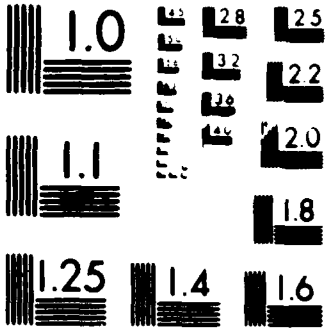

1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

Directory Reference Patterns in a UNIX Environment

Rick Floyd
Computer Science Department
The University of Rochester
Rochester, New York 14627

TR 179
August 1986

rochester
Department of Computer Science
University of Rochester
Rochester, New York 14627

12

Directory Reference Patterns in a UNIX Environment

Rick Floyd
Computer Science Department
The University of Rochester
Rochester, New York 14627

TR 179
August 1986

Abstract

Data on directory references made in opening files have been collected from a 4.2BSD UNIX system supporting university research. An analysis of these data shows that paths are relatively "long" (an average of 2.7 components to resolve per path) and that, in the absence of caching, name resolution overhead accounts for over 70% of the disk blocks referenced to open and use files. Directory references show strong locality, though, making caches an effective way to decrease this overhead. Simulations of an LRU whole directory cache show that a cache holding just 10 nodes achieves an 85% hit ratio.

A number of other results on directory reference patterns are presented in this paper, along with a discussion of their implications for both local and distributed file systems.

This work was supported in part by the National Science Foundation under grant number DCR-8320136 and in part by the Office of Naval Research under grant number N00014-82-K-0193.

A180023

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR 179	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Directory Reference Patterns in a UNIX Environment		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Rick Floyd		8. CONTRACT OR GRANT NUMBER(s) N00014-82-K-0193
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department The University of Rochester Rochester, New York 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA/1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE August 1986 13. NUMBER OF PAGES 36
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Directory, reference-patterns, UNIX file system, name resolution, Caching distributed file systems.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Data Directory references made in opening files have been collected from 4.2BSD UNIX system supporting university research. An analysis of these data shows that paths are relatively "long" (an average of 2.7 components to resolve per path) and that, in the absence of caching, name resolution overhead locality, though, making caches an effective way to decrease this overhead. Simulations of an LRU whole directory cache show that a cache holding just 10 nodes achieves an 85% hit ratio.

20. ABSTRACT (Continued)

A number of other results on directory reference patterns are presented in this paper, along with a ciscussion of their implications for both local and distrubuted file systems.

1. Introduction

File systems that provide a flexible hierarchical directory structure are rapidly becoming the norm rather than the exception. This is particularly true for distributed file systems (DFS's). Many recent DFS's [Ellis 83, Satyanarayanan 85, Tichy 84, Walker 83] use a name space modeled after the hierarchical UNIX[1] file system [Ritchie 78]. Understanding and improving the behavior of these hierarchically structured file systems has been hampered by a lack of information on the ways that they are used. In particular, there is very little data available on directory reference patterns.

Our recent study of short term file reference patterns on a 4.2BSD UNIX system [Floyd 86] found that referenced files tended to be small (under 1000 bytes long) and that some of the most commonly referenced files required as many as 4 path components to be resolved to reach the descriptor for the file. These two results suggest that even in a single site file system, name resolution can be an important factor in determining the overall performance of the file system. Reports that 40% of BSD UNIX system call overhead is due to name resolution [Leffler 84] support this. For a DFS, name resolution cost may be even more critical. If path components are resolved individually across the net, as they are in many DFS's, network overhead can make the cost prohibitively high.

There are a number of ways to decrease the name resolution cost. One of the simplest is to cache directory information. 4.3BSD UNIX caches individual directory entries. Sheltzer et al. [Sheltzer 86] have had good results with page level caching in the LOCUS distributed file system. Finally, entire directory nodes may be cached or replicated. Treating directories as a whole is the approach used by the Roe distributed file system [Ellis 83, Floyd 87] and is the one that we will be addressing here.

Our earlier paper describes modifications that were made to a local UNIX system to collect a log of accesses to files. This log does not capture all directory operations (for example, calls such as *lstat* that request information about a file but don't access file data are not logged). It does, however, include a complete record of the paths used to create, open, execute and delete files, to open directories, and to create, delete, and modify directories. This paper uses that path information to examine the overhead of name resolution in accessing files, the rate of change of directory nodes, the effectiveness of whole directory caches, and so on. We have generally tried to present the information in a way that gives a qualitative feel for the way that directories are used on our UNIX system. Quantitative fits and distributions are, for the most part, sacrificed in favor of observations that would aid in developing and operating file systems. These results, along with our earlier file reference study and a simulation driven by the data we have collected, will be used to investigate the performance of the Roe distributed file system.

Section two of this paper describes the environment in which our measurements were made and presents a brief overview of the data collection method. Section 3 outlines the approach we used in analyzing the data. Section 4 presents some of the major results of this analysis. In section 5, we discuss the implications that these results have for file system design. Section 6 describes further analysis that could be done and section 7 summarizes our results.

Familiarity with UNIX [Ritchie 78] is assumed. Knowledge of 4.2BSD UNIX [Joy 83] may also be useful.

[1] UNIX is a trademark of AT&T Bell Laboratories.

2. Data Collection

2.1. Data Collection Environment

The data used in this paper were collected from a VAX 11/780[2] on the University of Rochester Computer Science Department network. At the time that the data was collected (September 1985), the network consisted of a VAX 11/780, 4 VAX 11/750's, 7 Sun workstations, 13 Xerox Dandelion workstations, 3 Symbolics LISP machines, and a number of special purpose devices. The 11/780, called Seneca, was selected as the primary machine for data collection because it was far and away the most heavily used of our systems. Seneca had, at the time, 4MB of memory, 560MB of disk storage and was running 4.2BSD UNIX. The system supported roughly 200 users. The primary user activities were program development (as part of our research effort), text editing and formatting, reading news and reading personal mail. Seneca also acted as a USENET news and UUCP mail relay [Nowitz 78]. There was relatively little database activity.

Data were also collected from two of the 11/750's. Preliminary analysis of the 11/750 data merely confirmed the importance of Seneca in our environment. Neither of the 11/750's had file system activity levels greater than 15% of that seen on Seneca. Because of this, only the Seneca data were fully analyzed.

2.2. Data Collection Method

In this section we briefly describe the data collection package. A more detailed description may be found in [Floyd 86]. When we refer to "the file system" in this section (and in the rest of the paper), we will generally be considering the overall file name space on Seneca. On UNIX machines, the file name space is actually made up of a number of physical file systems, each of which stores files, directories, descriptors, and device information for a subtree of the overall naming tree. If we are discussing a particular physical file system, it will be mentioned explicitly.

Two types of data were collected: 1) a static "snapshot" of the file system and 2) a running log of file system activity.

The static snapshot provides a picture of the entire file structure on a machine at a given point in time. The snapshot includes information on the name, file id (device/inode), and size of files. It also includes the name, file id, and children of directories. A static snapshot was taken of the Seneca file system when file system logging was started. This snapshot was used as a starting point for the analysis programs (section 3) and also provided information on the static directory size distribution.

The 4.2BSD UNIX kernel was modified to log system calls that accessed file and directory data or modified directories. Logged calls that resulted in directory references can be classified as follows:

(1) Directory structure modifications: mkdir, rename, rmdir, symlink.

(2) File data references: execv/execve, link, open/creat, truncate, unlink.

(3) Process context: chdir, chroot.

[2] VAX is a trademark of Digital Equipment Corporation.

The first four records (mkdir, rename, rmdir and symlink), combined with the results of the static snapshot taken at the start of logging, allow us to construct and maintain a model of the directo:y tree for the file systems on the machine. *Mkdir* creates a new directory. *Rename* changes the path used to reach an object. *Rmdir* deletes a directory. *Symlink* creates a symbolic link containing a path to a file or directory. When a symbolic link is encountered during path resolution, the path in the symbolic link is substituted into the partially resolved path before resolution is continued. This is the only way in 4.2BSD UNIX to make links across file systems.

Calls in the second category (execute, link, open, truncate, and unlink) are the actual references to files. *Execute* (execv and execve system calls) executes a file, replacing the current process with the image given in the file. *Link* and *unlink* add and delete directory entries for files. While these calls don't refer to file data explicitly, if unlink removes the last directory entry for a file, the file will be deleted. *Open* (open and creat system calls) opens or creates a file or opens a directory. Processes access files either by explicitly opening them or by inheriting open files from their parents. *Truncate* shortens a file.

The remaining records (chdir and chroot), along with information on process creation and destruction (not shown), allow us to keep track of the location in the directory tree of each process. *Chdir* changes the directory used to resolve relative references made by a process (those not starting from the root of the file system tree). *Chroot* changes the root of the file system as seen by a process.

Each log record includes the time that the call finished (with a resolution of 10ms), the process identifier (pid) of the process making the request, and information describing the call arguments and result.

Our original purpose in collecting data was to track references to files. Because of this, some calls that cause directories to be referenced (for name resolution) were omitted from the log. These calls were:

(1) Protection: chmod, chown.

(2) Status: readlink, lstat, stat, utimes, access.

(3) Administrative: acct, mknod, mount, setquota.

(4) UNIX domain IPC (side effect of the 4.2BSD implementation): bind, connect.

Of these calls, only *lstat* and *stat* are likely to occur with any frequency. These two calls retrieve status information on a file (size, protection, access date, and so on) from the file inode. Mogul found [Mogul 86b], in studying another 4.2BSD system, that lstat and stat were used nearly twice as often as the calls that we logged. Most of these status calls were made, though, by an administrative process that scanned the entire file system on a regular basis. A similar program is run on our system, but only half as often and on a smaller, busier file system. Based on this, we estimate that lstat and stat calls occur about half as often as open and creat calls on Seneca. Further, these status calls are generally tightly clustered in time (most will occur during the 4AM scan of the file system) and so we expect that they will have little effect on the results we will be presenting.

Another potential contribution to directory references that we have not logged is from system calls that fail. We logged only successful calls.

3. Analysis Method

3.1. Basic Approach

To analyze a file reference log, we first set up a model of the original file system using a snapshot taken when logging was enabled. We can then interpret log records in the context of this model, updating the model as necessary. This approach gives us information such as directory sizes, owner, and reference history that is not available from the raw log (see [Floyd 86] for more on this).

When we encounter a record in the log that contains a path to resolve, we take each path component in turn, resolving it individually. Each directory used in the resolution is marked as having been referenced and the appropriate histograms are incremented. Resolution starts either at the root of the file system (for an absolute path) or in the current working directory of the process that generated the record. So, for example, an OPEN record that specifies a path of "/u/rick/.login" generates 3 directory references: to "/" to resolve "u"; to "u" to resolve "rick"; and to "rick" to resolve ".login."

There are a few conventions that are used in the analysis:

(1) All of the analysis presented here is at the *node* (entire directory) level. We haven't attempted to analyze references to individual directory entries or pages. See [Floyd 86] (particularly the file interopen interval and lifetime distributions) and [Leffler 84] for information on individual entries.

(2) Directory sizes include entries for "." (the directory itself) and ".." (the parent). These entries are always present in a UNIX directory and so the minimum directory size is 2 entries.

(3) Directory sizes given in bytes or blocks assume that the 4.2BSD directory layout is used (that is, an 8 byte header, space for the name itself, and a 1 byte trailer, padded out to a 4 byte boundary) and that there are no "empty" entries. This last assumption means that we probably understate somewhat the number of blocks required to read a directory.

(4) All component resolutions are marked as having taken place at the time the system call being analyzed finished. In real life, of course, these resolutions won't occur simultaneously. Because of this, intervals of less than 100ms should not be taken seriously.

(5) All components in a path are resolved. No attempt is made to short-circuit degenerate components or path segments. So, for example, resolving "./.login" requires two references (one for "." and one for ".login") and resolving "/u/rick/.login" would require 3 references even if the working directory of the process making the request is "/u/rick." This is consistent with the approach used by 4.2BSD.

3.2. Cuts

One expects that directory reference patterns will be different for user vs. system processes, user vs. system directories, batch vs. interactive work, and so on. This is certainly the case with file references [Floyd 86]. We have developed a set of data cuts that allows us to isolate and compare various contributions to the overall activity on Seneca. We use 4 basic types of cuts:

(1) Cuts on the ruid (owner) of processes making requests (the categories are UUCP/USENET network, system, and user).

(2) Cuts on the owner of the referenced directory (UUCP/USENET network, system, and user).

(3) Cuts on the owner of the referenced object (UUCP/USENET network, system, and user).

(4) Cuts on the *UNIX file system* of the referenced directory. As we mentioned in section 2.2, the overall file name space on a UNIX machine is actually made up of a number of physical file systems. There were 5 physical file systems on Seneca at the time data was collected, mounted as follows: "/," "/u," "/usr," "/usr/spool," and "/tmp".

Some of these cuts may be combined to give other more specific cuts. 9 cuts are used in this paper:

(1) **no cut**: This cut passes all records in the log to the user analysis routines.

(2) **ruid_NET**: Passes references by what we term *net* processes. Net processes are those running under UUCP, USENET news or notes accounts. Most of these processes run in batch mode and so this cut gives us a sample that is considerably different from an interactive one. This category has been broken out from the system and user categories because of the batch-oriented nature of the references and the large number of system calls by net processes (roughly 1/3 of the calls in this study and as much as 70% of the non-system calls in earlier studies [Floyd 85]). We don't include activity due to Seneca being on the Rochester network in the ruid_NET category.

(3) **ruid_SYSTEM**: Passes references by system processes (those running under root, daemon, games and other miscellaneous system accounts). System processes are primarily daemons that provide widely used services (such as spooling and network status reporting), processes created on behalf of users to perform privileged operations, and periodic maintenance processes.

(4) **ruid_USER**: Passes references by processes running under user accounts.

(5) **dir_owner_NET**: Passes references to directories owned by UUCP, USENET news, and notes accounts. These are primarily directories holding news articles, UUCP spool files, and news and UUCP activity logs.

(6) **dir_owner_SYSTEM**: Passes references to directories owned by the system accounts mentioned above. This includes directories holding major administrative and status files (for example, /etc), system libraries, system include files, the root of each mounted file system, and so on.

(7) **dir_owner_USER**: Passes references to user directories.

(8) **owner_USER+ruid_USER**: Passes references made by user processes, but only if the leaf object is owned by a user. This gives a trace of directories accessed in resolving user references to user files. The owner_USER+ruid_USER cut is interesting because it gives us a measure of activity that is relatively independent of the underlying system.

(9) **ruid_USER+/u**: Passes references made by user processes to directories in the /u file system (this is the file system on Seneca that holds all user directories). While the owner_USER+ruid_USER cut includes all directories that are referenced in accessing user files (including, for example, "/tmp" for temp files and "/" for absolute path names), the ruid_USER+/u cut only includes that subset of files and directories on /u. This cut will be of particular interest to DFS designers who combine a global user file space with local system directories [Satyanarayanan 85].

4. Directory Reference Patterns

This section presents the results of our analysis. A full analysis of the data was done using 21 different cuts (the 9 cuts listed in section 3.2 plus cuts on other file systems and on ruid, file system, and directory owner combinations). It is clearly impractical to present results for the full set of cuts. We have generally included only those tables and histograms that are particularly characteristic or striking.

Overall system activity and per call results are presented using ruid cuts. These cuts show the overall contribution from each of the user classes and point out some of the differences in the way that these classes use the file system. Analysis that concentrates on individual directories is presented using directory owner cuts. Directory owner cuts show us roughly where in the file system activity is concentrated and allow us to investigate the activity on a directory by directory basis.

In some cases we give more detailed results on user activity using the owner_USER + ruid_USER and ruid_USER + /u cuts. These cuts give us a data sample that allows us to investigate reference patterns that a DFS dealing primarily or wholly with user files would see (Roe [Ellis 83] and the ITC DFS [Satyanarayanan 85] are examples of such a DFS).

4.1. Basic Statistics

Roughly 7 days of data were collected on Seneca (168.82 hours, from 3:21AM on Monday, September 16, 1985 to 4:10AM on Monday, September 23). During this period there were 142 active users of the system. There were generally 20 to 30 logged in users at any given time on weekday afternoons, with load averages running between 5 and 10.

A summary of the records collected that reference directories is given in table 1. The first 3 columns give the number of records of each type collected, the average rate for that type of record, and the percentage of

record	no cut			ruid_NET		ruid_SYSTEM		ruid_USER	
	count	per hr	fraction	count	fraction	count	fraction	count	fraction
mkdir	936	5.5	0.07%	795	0.19%	2	0%	139	0.03%
rename	3211	19	0.23%	1946	0.46%	408	0.08%	857	0.19%
rmdir	913	5.4	0.06%	780	0.19%	0	-	133	0.03%
symlink	16	0.1	0%	0	-	3	0%	13	0%
chdir	136063	806	9.7%	19102	4.5%	71854	13.5%	45106	10.0%
chroot	0	-	-	0	-	0	-	0	-
execute	125064	741	9.0%	26761	6.4%	38093	7.2%	60209	13.3%
link	42929	254	3.1%	25694	6.1%	7301	1.4%	9934	2.2%
open	965087	5720	68.7%	277350	65.9%	393661	74.1%	294070	64.9%
truncate	0	-	-	0	-	0	-	0	-
unlink	130929	776	9.3%	68342	16.2%	19861	3.7%	42726	9.4%
total	1405148	8323	100%	420770	100%	531183	100%	453187	100%

Table 1: Records logged

collected records that this represents. The remaining columns show the number of records collected cut by the ruid of the calling process and the percentage of the total for the ruid class. Opens accounted for 2/3 of the path requests we logged. Chdir, unlink, and execute calls accounted for most of the rest of the requests. There were relatively few directory structure modification requests.

Opens may be further broken down by the type of object being opened (table 2). While most requests were to open regular files, there were also a significant number of directory opens. Processes open directories in UNIX to scan the contents (as opposed to resolving a single name). This is commonly done by user processes to satisfy interactive requests to list directory contents. Directory open activity by system processes was due to daemons examining spool directories for work and to housekeeping scans of the file system. In our analysis we have counted the open of a directory as a single reference to the directory.

Each of our ruid classes accounted for roughly 1/3 of the path resolution requests (table 3). Most of these paths were specified absolutely (that is, were resolved starting at the root of the naming tree). Overall, only about a quarter of the objects being referenced were listed in the working directory of the process making the request. This is a reflection, in part, of the high level of activity to system files. As we saw in [Floyd 86], over half of all file opens went to system files. 4.2BSD makes heavy use of system files to store system configuration and status information. Since these files are often opened as an incidental part of other activity, they are usually not the current working directory and so are referenced absolutely.

Our compound user cuts eliminate this activity to system files. If we look at just user activity to files on /u (the user file system), we find that 2/3 of the paths we saw specified an object in the working directory of the process making the request. Note, though, that user references to objects on /u accounted for only a third of the overall user paths and a tenth of the system activity. For references by users to all user objects

type	no cut		ruid_NET		ruid_SYSTEM		ruid_USER	
	opens	fraction	opens	fraction	opens	fraction	opens	fraction
regular file	754285	78.2%	249825	90.1%	298186	75.7%	206268	70.1%
directory	170448	17.7%	17275	6.2%	72625	18.4%	80548	27.4%
block special	922	0.1%	0	-	60	0.02%	862	0.3%
character special	39432	4.1%	10250	3.7%	22790	5.8%	6392	2.2%
total	965087	100%	277350	100%	393661	100%	294070	100%

Table 2: Opens, by object type

cut	paths	% paths	absolute path	leaf in working dir
ruid_NET	4.48e5	30.8%	74.2%	17.1%
ruid_SYSTEM	5.39e5	37.0%	70.9%	28.3%
ruid_USER	4.69e5	32.2%	66.6%	32.8%
owner_USER + ruid_USER	2.55e5	17.5%	54.3%	45.9%
ruid_USER + /u	1.57e5	10.8%	35.2%	64.7%
no cut	1.46e6	100%	70.5%	25.9%

Table 3: Path statistics

(those on /u plus user files in shared system directories such as /tmp and /usr/spool/mail), the fraction of paths specifying an object in the working directory drops to less than 1/2.

Each path resolved may (and usually does) have more than one component. Table 4 gives some information on the number of components per path for each of our ruid classes. Note that each path had, on average, almost 3 components, each of which required a directory reference to resolve. Paths for net processes were particularly long. This was caused by the relative depth of the net directory trees (rooted in /usr/spool/news, /usr/spool/uucp, and so on) coupled with the heavy use of absolute path names by net processes.

User paths specifying user objects were generally shorter, with an average of slightly more than two name components to resolve per path. If the target object was on /u, an average of 1.57 of the components resolved were on the /u file system (note that this doesn't include references to "/" for absolute paths, since "/" is not on the /u file system).

While each of the ruid categories accounted for a roughly equal number of references, nearly 3/4 of all references went to system directories (table 5). This is not surprising, since most references were absolute

cut	mean	median	1 component	2	3	4	>4	max
ruid_NET	3.45	4	8.0%	34.9%	6.3%	25.7%	25.2%	8
ruid_SYSTEM	2.48	2	28.4%	31.0%	6.3%	33.5%	0.7%	8
ruid_USER	2.22	2	32.5%	34.7%	19.2%	8.9%	4.7%	11
owner_USER + ruid_USER	2.11	2	42.2%	25.3%	18.2%	10.5%	3.8%	11
ruid_USER + /u	1.57	1	61.9%	26.7%	6.5%	2.7%	2.1%	9
no cut	2.70	2	23.4%	33.4%	10.5%	23.2%	9.5%	11

Table 4: Components/path

cut	total		reads		writes		reads/
	references	fraction	references	fraction	references	fraction	writes
ruid_NET	1.59e6	37.5%	1.44e6	36.4%	1.42e5	52.1%	10.1
ruid_SYSTEM	1.48e6	34.9%	1.44e6	36.4%	4.03e4	14.8%	35.7
ruid_USER	1.17e6	27.6%	1.08e6	27.2%	9.01e4	33.1%	12.0
dir_owner_NET	7.42e5	17.5%	6.17e5	15.6%	1.25e5	46.1%	4.9
dir_owner_SYSTEM	3.09e6	73.0%	2.96e6	74.7%	1.23e5	45.4%	24.1
dir_owner_USER	4.06e5	9.6%	3.83e5	9.7%	2.32e4	8.6%	16.5
owner_USER + ruid_USER	6.16e5	14.5%	5.36e5	13.5%	7.99e4	29.4%	6.7
ruid_USER + /u	3.07e5	7.2%	2.86e5	7.2%	2.20e4	8.1%	13
no cut	4.24e6	100%	3.96e6	100%	2.72e5	100%	14.6

Table 5: Reference statistics

and so this implies that even references to files in user or news subtrees often required two or three system directories to resolve. There was relatively little activity to user directories. Overall, 93.4% of the references were directory reads and 6.6% were directory writes[3]. Nearly half of the writes were to system directories (mostly to /tmp and /usr/spool/mqueue), with most of the rest going to net directories. Net directories were not heavily used overall, but had a particularly low read/write ratio and so a high fraction of the writes.

User references to their files and directories accounted for only 14.5% of the references on the system and about half of the references made by users. Relatively few user writes were to user directories. As we will see in section 4.3, most were to system temporary and spool directories.

Most directories belonged to users (table 6) but, as we saw above, there was relatively little activity to these directories. Again, this is a reflection of the heavy activity to system directories.

cut	directories	% directories	references/directory
dir_owner_NET	1275	23.5%	582
dir_owner_SYSTEM	427	7.9%	7230
dir_owner_USER	3713	68.6%	109
no cut	5415	100%	782

Table 6: References/directory

4.2. Per Reference Results

The directory reference activity over time is shown in figure 1. References followed a daily pattern with a busy period between 9am and 6pm, overlaid by bursts from net activity (news reception) and a strong peak in the early morning (news expiration and the housekeeping scan of the file system). Weekends were relatively quiet. Except for the strength of the early morning peak, this pattern follows closely the one we saw for file opens [Floyd 86]. The relative strength of the morning peak is due to the long length of paths used by net processes and the inclusion of directory opens (the primary housekeeping activity we logged).

User activity to user files (figure 2) showed a busy period during the day, with activity tapering off in the late evening. This is typical of a university environment. There was some early morning activity due to user background jobs.

[3] Note that directory writes in UNIX represent changes in the naming tree (such as adding and deleting files) Information on the objects named (size, last use and so on) is kept elsewhere

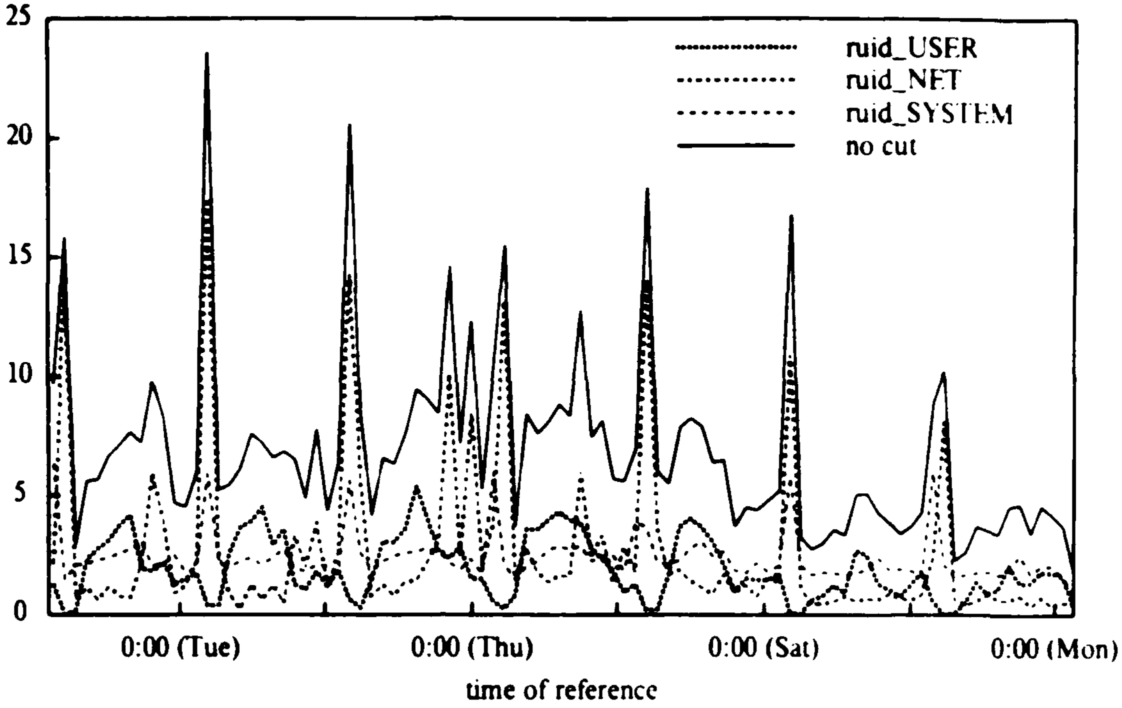


Figure 1: Directory references per second (˜2 hour resolution)

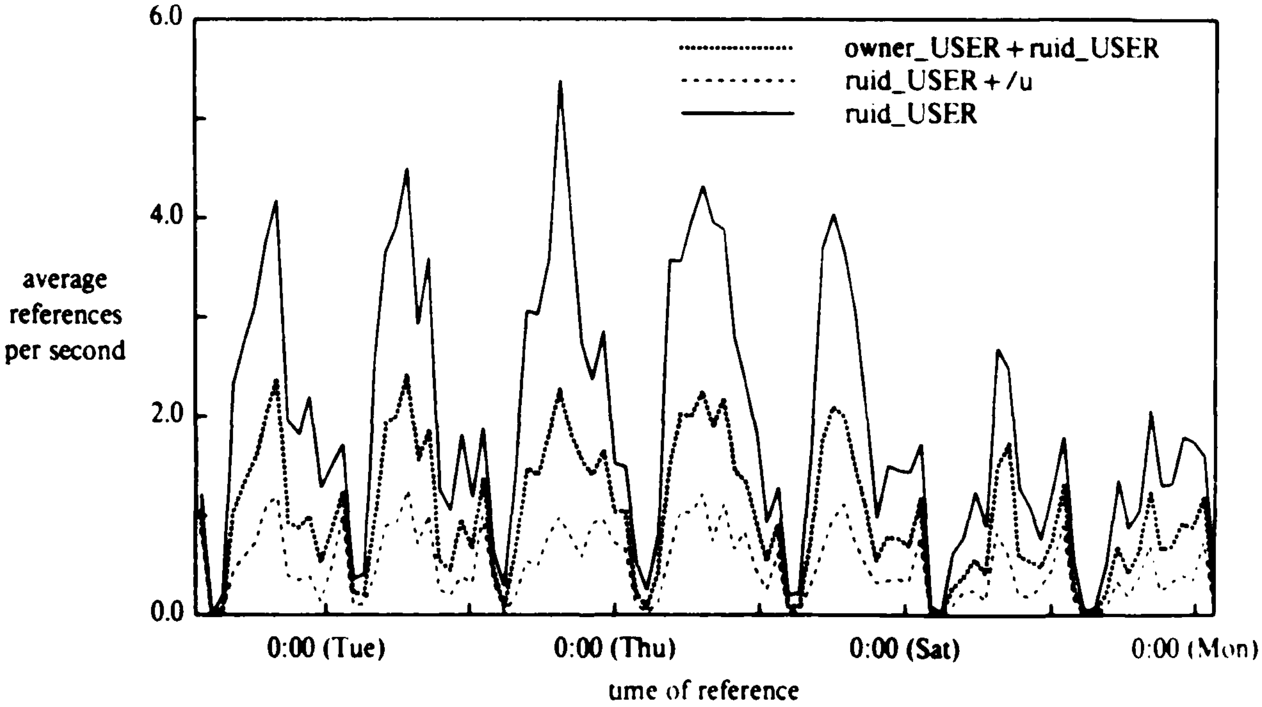


Figure 2: Directory references per second (˜2 hour resolution, user cuts)

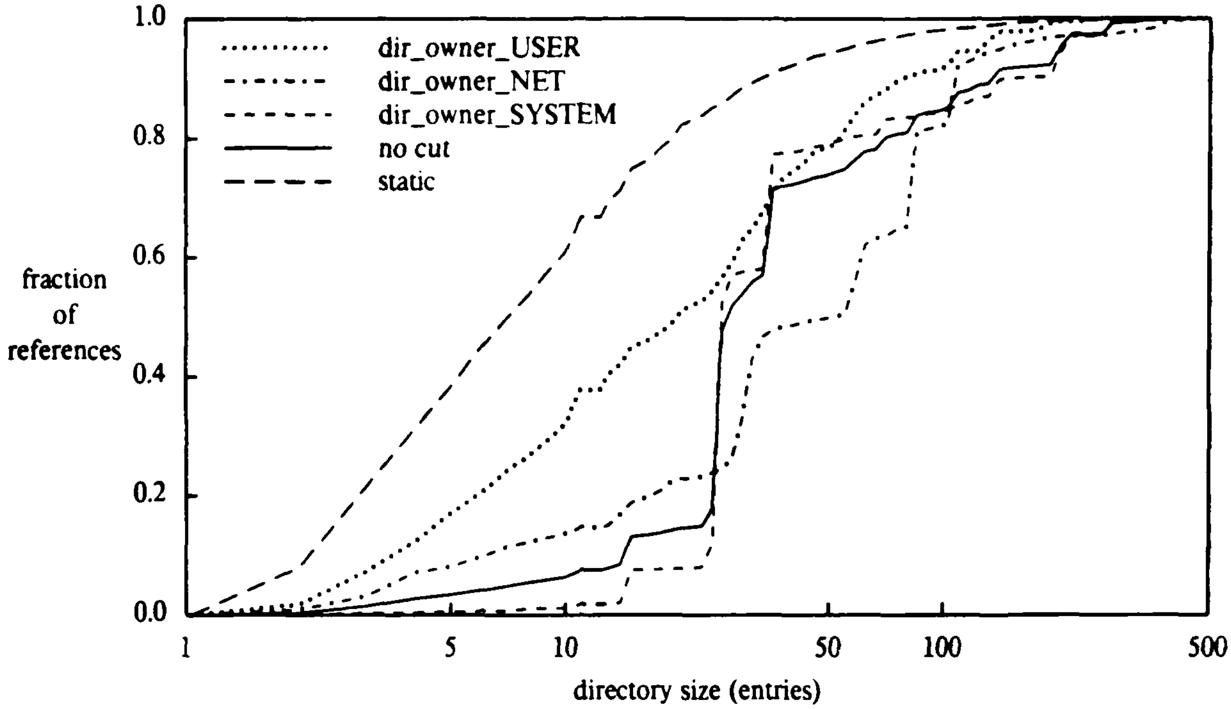


Figure 3: Size of referenced directories (cumulative, in entries)

distribution	min	max	mean	median	std deviation
dir_owner_NET. dynamic	2	500	63.1	51	65.3
dir_owner_SYSTEM. dynamic	2	471	55.1	26	63.3
dir_owner_USER. dynamic	2	327	34.2	20	40.2
owner_USER + ruid_USER	2	484	60.8	29	64.2
owner_USER + /u	2	282	82.9	40	80.6
ruid_USER	2	484	66.0	33	69.8
all, dynamic	2	500	54.5	26	62.2
all, static	2	471	15.8	8	27.3

Table 7: Directory size distributions (in entries)

Figure 3 shows the size (in entries) of referenced directories, weighted by the number of references made and cut by the owner of the referenced directory. Note that these are *cumulative* distributions. At any point on a curve, the y value is the fraction of directories with sizes less than or equal to the x value. For comparison purposes, we have included here the static directory size distribution (this is the distribution that would result if each directory on the system were referenced once). Table 7 gives some statistics on these distributions.

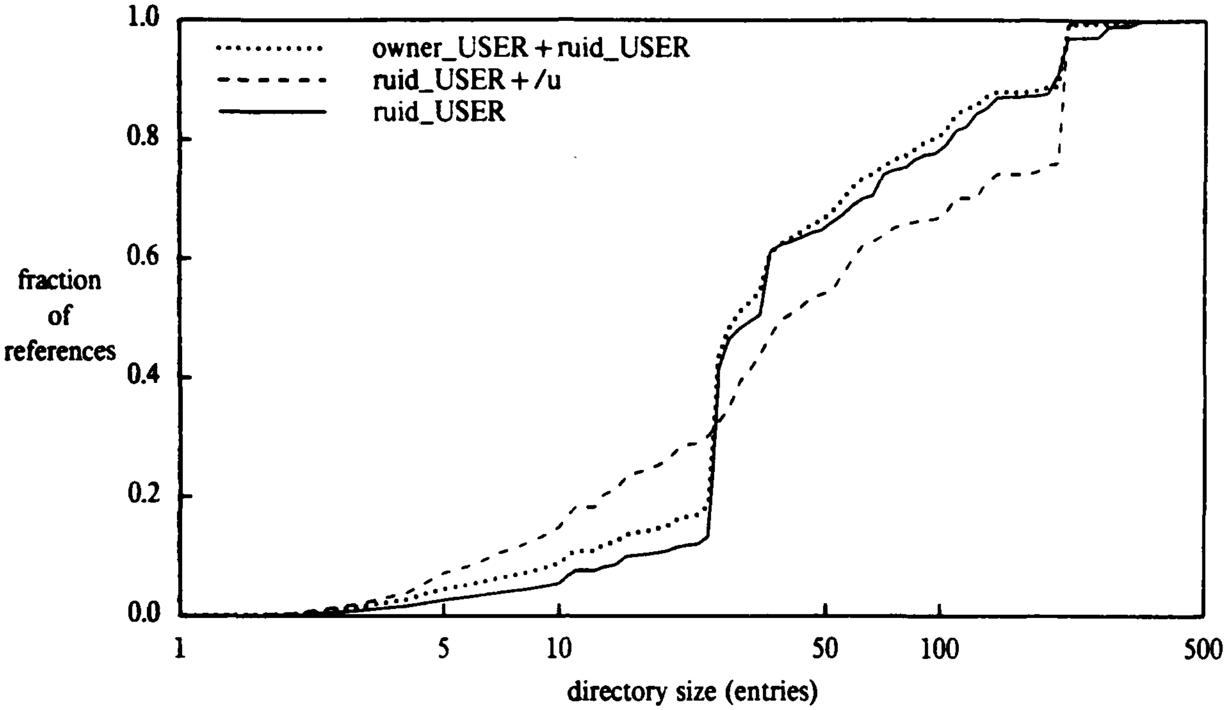


Figure 4: Size of referenced directories (cumulative, in entries, user cuts)

From figure 3 we see that most directories on Seneca were small (half had under 8 entries). Referenced directories were considerably larger (median of 26 entries), but still small by most standards. Since "/," "/usr," and "/usr/spool" had 26, 35 and 26 entries respectively and accounted for nearly half of the references, this result was inevitable. The median size of 26 entries implies that, in the absence of other factors, the median number of comparisons needed to resolve a component was 13. This agrees with 4.2BSD measurements done elsewhere [Mogul 86a]. The small static median is also typical of 4.2BSD systems [Mogul 86b]. These distributions all have long tails and so the means are considerably higher.

Directories on /u referenced by users (weighted by the number of references) were generally somewhat larger than referenced directories on the system as a whole (figure 4 and table 7). This was due, in part, to the relatively high level of activity to /u (at 200 entries) and the absence of the heavily referenced system directories (at 26 entries).

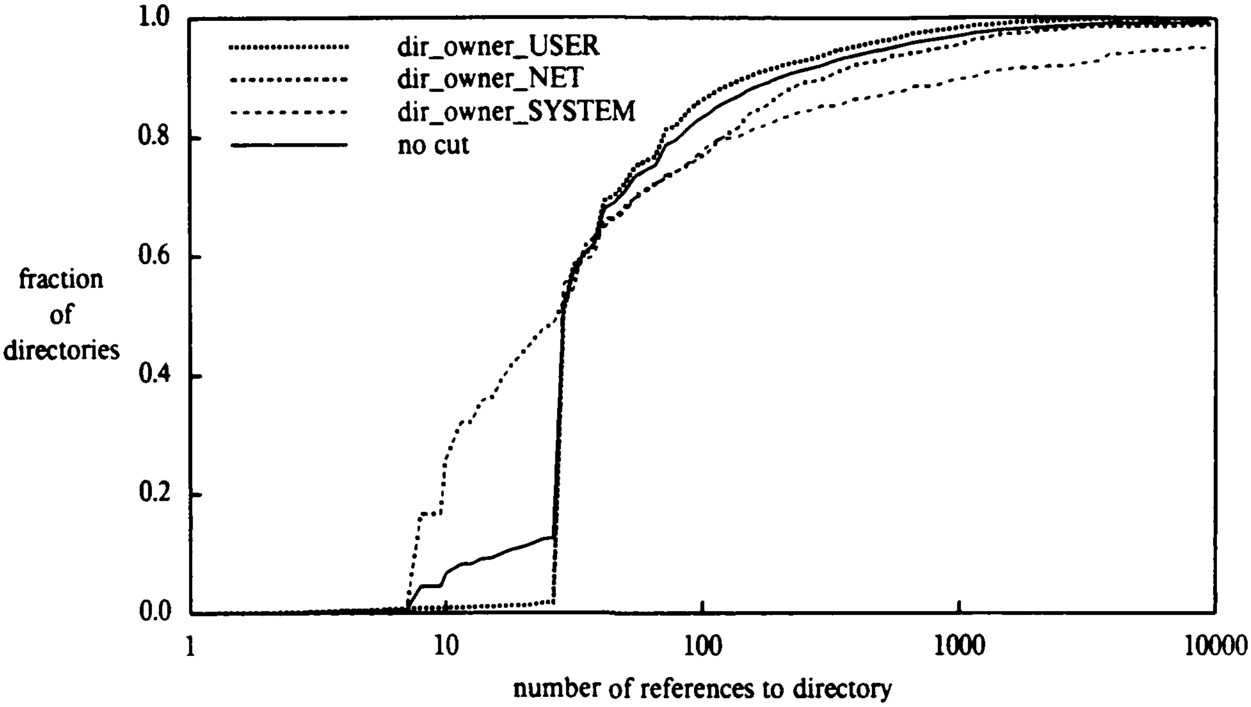


Figure 5: Number of references per directory (cumulative)

distribution	min	max	mean	median	std deviation
dir_owner_NET	8	1.45e5	582	28	5.9e3
dir_owner_SYSTEM	28	1.06e6	7230	28	6.3e4
dir_owner_USER	2	1.38e4	109	28	4.1e2
owner_USER + ruid_USER	1	1.44e5	327	12	4.0e3
ruid_USER + /u	1	7.18e4	219	18	2.0e3
ruid_USER	1	3.24e5	582	20	8.1e3
no cut	2	1.06e6	782	28	1.8e4

Table 8: Number of references/directory

4.3. Per Directory Results

The number of references to a directory over a period gives an indication of the potential benefits of caching or, for a DFS, of migrating or replicating a directory (update activity and sharing are also important factors). If we ignore scans of the entire file system (at least 28 references per directory over the course of the week) we see that half of the directories on the system were not referenced at all (figure 5 and table 8). Many of the rest were referenced a few tens of times.

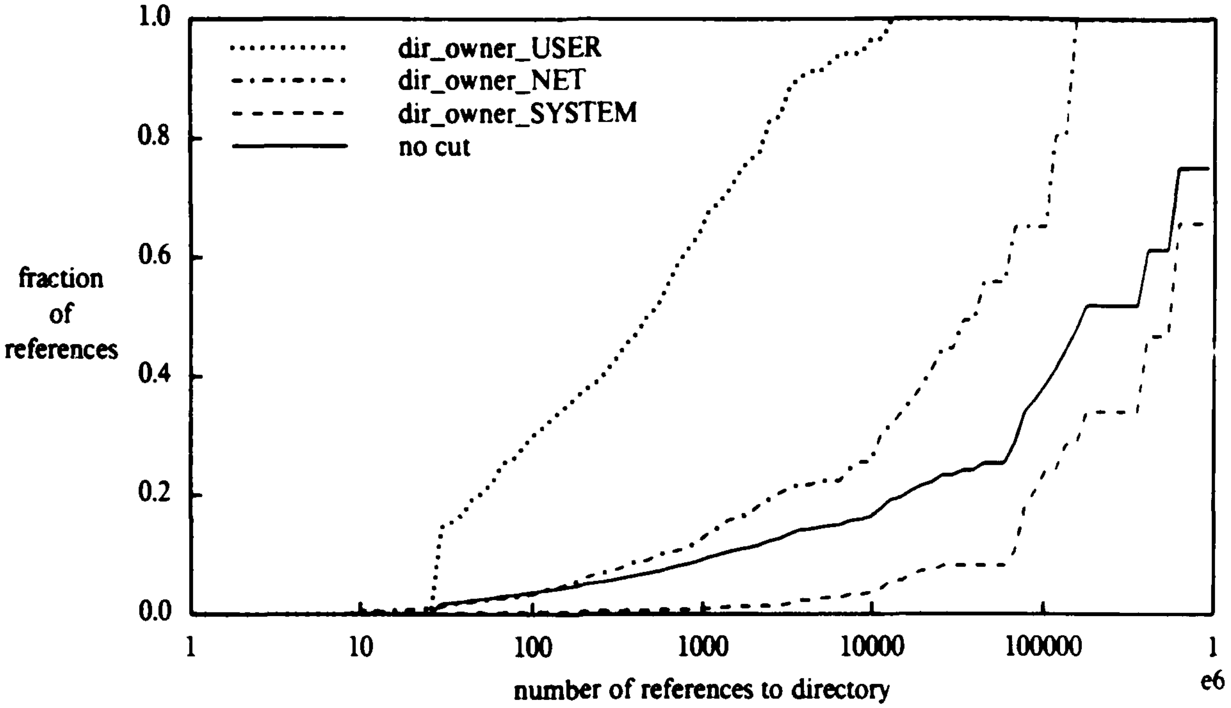


Figure 6: Fraction of references per active directory (cumulative)

distribution	mean	median	std dev
dir_owner_NET	6.09e4	4.47e4	5.6e4
dir_owner_SYSTEM	5.54e5	6.17e5	4.1e5
dir_owner_USER	1.67e3	4.68e2	2.9e3
no cut	4.15e5	1.78e5	4.2e5

Table 9: Reference distribution (as a function of references/directory)

There were some net and system directories, though, that were referenced tens of thousands of times. Over half of the references, in fact, went to system directories referenced more than 100,000 times each. This is shown in figure 6 and table 9, where we have weighted the distributions in figure 5 by the number of references made. This gives us the fraction of overall references as a function of directory activity. Note that 85% of the references went to directories referenced more than 10,000 times.

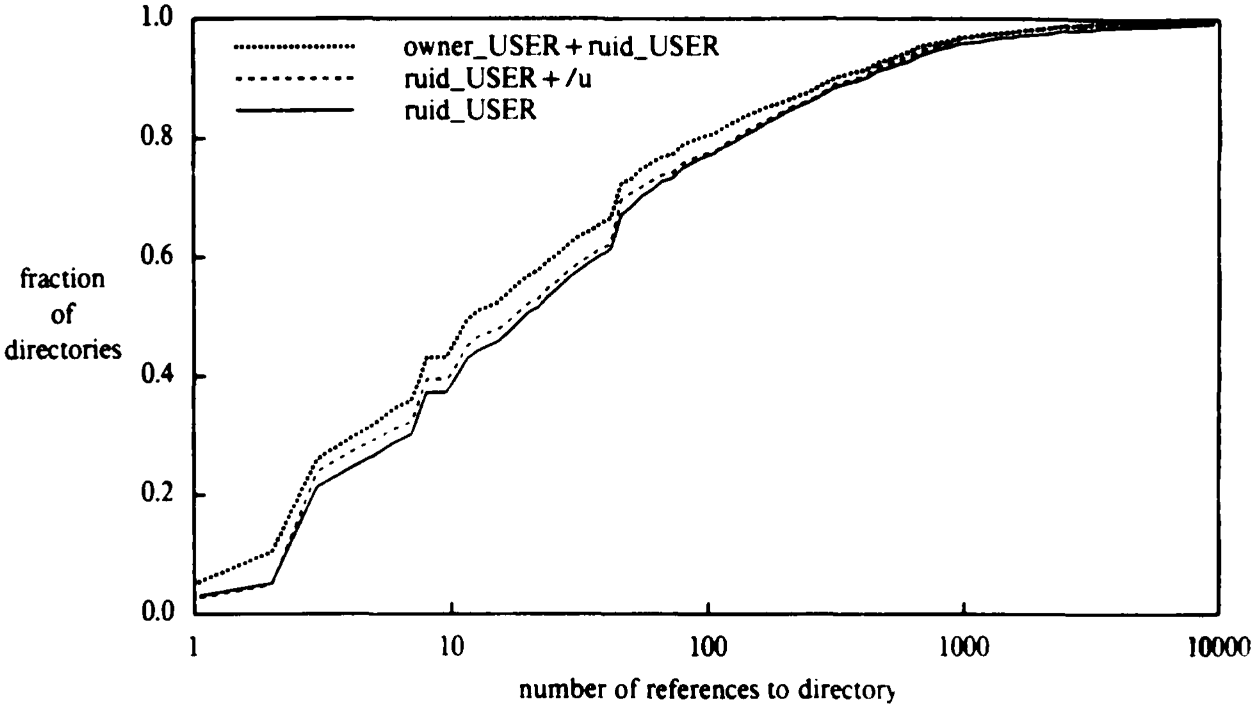


Figure 7: Number of references per active directory (cumulative, user cuts)

Figure 7 and table 8 show, for each of the user cuts, the number of references made to *active* directories (those actually referenced given the cuts). If a directory was referenced at all by users (only 37% were), it was likely to see enough activity to make trying to minimize the access overhead (through caching, migration, or other mechanisms) worthwhile.

The most frequently referenced directories are listed in table 10. Note that the four busiest directories accounted for over half of the references and received, between them, just 9 writes in a week. These directories are clearly very good candidates for extensive replication in a DFS, since update overhead is not an issue. The 15 most active directories accounted for 76% of the references. This suggests that even in a local environment, special treatment of a small number of directories could result in substantial improvements in name resolution performance.

The directories most frequently referenced by users in accessing their files and data are listed in table 11. Most are shared user directories.

references	fraction	reads	writes	reads/writes	path
1058380	25.0%	1058374	6	176400	/
587013	13.9%	587013	0	-	/usr
395132	9.3%	395129	3	131700	/usr/spool
168205	4.0%	168205	0	-	/usr/spool/rwho
145318	3.4%	118865	26453	4.49	/usr/spool/uucp
136363	3.2%	136121	242	562	/etc
114227	2.7%	104485	9742	10.7	/usr/spool/news
105239	2.5%	51857	53382	0.97	/tmp
85579	2.0%	85566	13	6580	/usr/lib
78696	1.9%	78696	0	-	/u
76778	1.8%	22820	53958	0.42	/usr/spool/mqueue
74590	1.8%	74590	0	-	/bin
71037	1.7%	71037	0	-	/dev
69093	1.6%	69093	0	-	/usr/spool/news/net
47737	1.1%	38770	8967	4.32	/usr/lib/news
35091	0.82%	35085	6	5800	/usr/spool/notes1.nyu

Table 10: Frequently referenced directories (no cut)

references	fraction	reads	writes	reads/writes	path
143956	23.4%	143954	2	72000	/
63235	10.3%	63235	0	-	/u
58955	9.6%	22917	36038	0.64	/tmp
27744	4.5%	27744	0	-	/usr
20988	3.4%	20988	0	-	/usr/spool
14836	2.4%	4614	10222	0.45	/usr/spool/mqueue
13138	2.1%	11770	1368	8.6	/u/ken
12286	2.0%	5521	6765	0.82	/usr/spool/mail
8967	1.5%	8963	4	2240	/u/ken/Src
6358	1.0%	4712	1646	2.86	/usr/spool/uucp
5980	0.97%	5336	644	8.29	/u/goddard/c400/assg2/coding
5328	0.86%	5233	95	55	/u/lee
5159	0.84%	4823	336	14.4	/usr/spool/news
3665	0.59%	3456	209	16.5	/u/scott/src/window
3510	0.57%	3510	0	-	/usr/local

Table 11: Frequently referenced directories (owner_USER + ruid_USER cut)

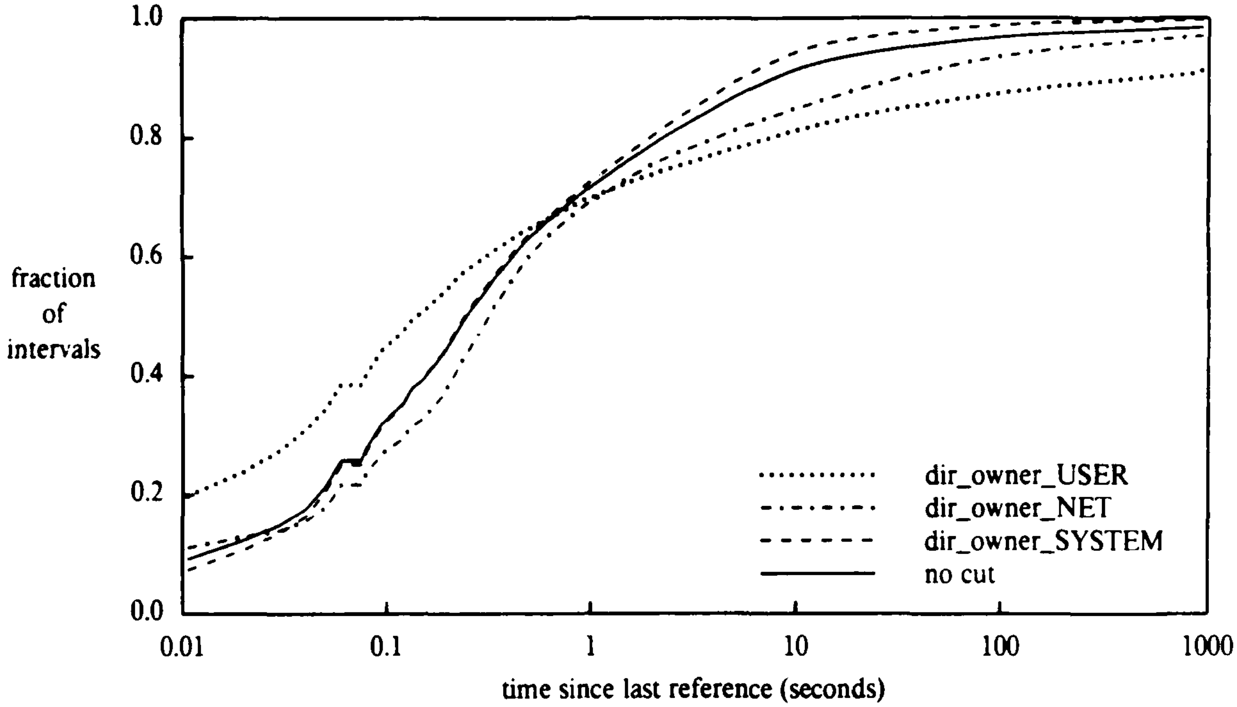


Figure 8: Directory inter-reference intervals (cumulative)

distribution	min	max	mean	median	std dev
dir_owner_NET	0	8.7e4	351	0.33	4.1e3
dir_owner_SYSTEM	0	8.7e4	73	0.27	2.2e3
dir_owner_USER	0	8.7e4	4680	0.15	1.8e4
no cut	0	8.7e4	560	0.27	6.4e3

Table 12: Directory inter-reference intervals (seconds)

Knowledge of directory inter-reference intervals (the time from one reference of a directory to the next) is useful in estimating both the appropriate time scale for migration and the possibilities for caching. Figure 8 and table 12 show that inter-reference intervals were short (opens to directories were strongly clustered). When a directory was referenced, the following reference (if any) had a 50% probability of occurring in the next 1/4 second. Part of this may be attributed to the heavily used system directories, but net and user directories also had short median inter-reference times[4]. There is strong reference locality in both time and space.

[4] The large fraction of zero length intervals for user directories was due to redundant references and opens to the current working directory, and to routines such as getwd that find the path of the current working directory by traversing up the directory tree to the root and then back down. That these are all binned at zero is partly an artifact of our analysis technique (see section 3.1)

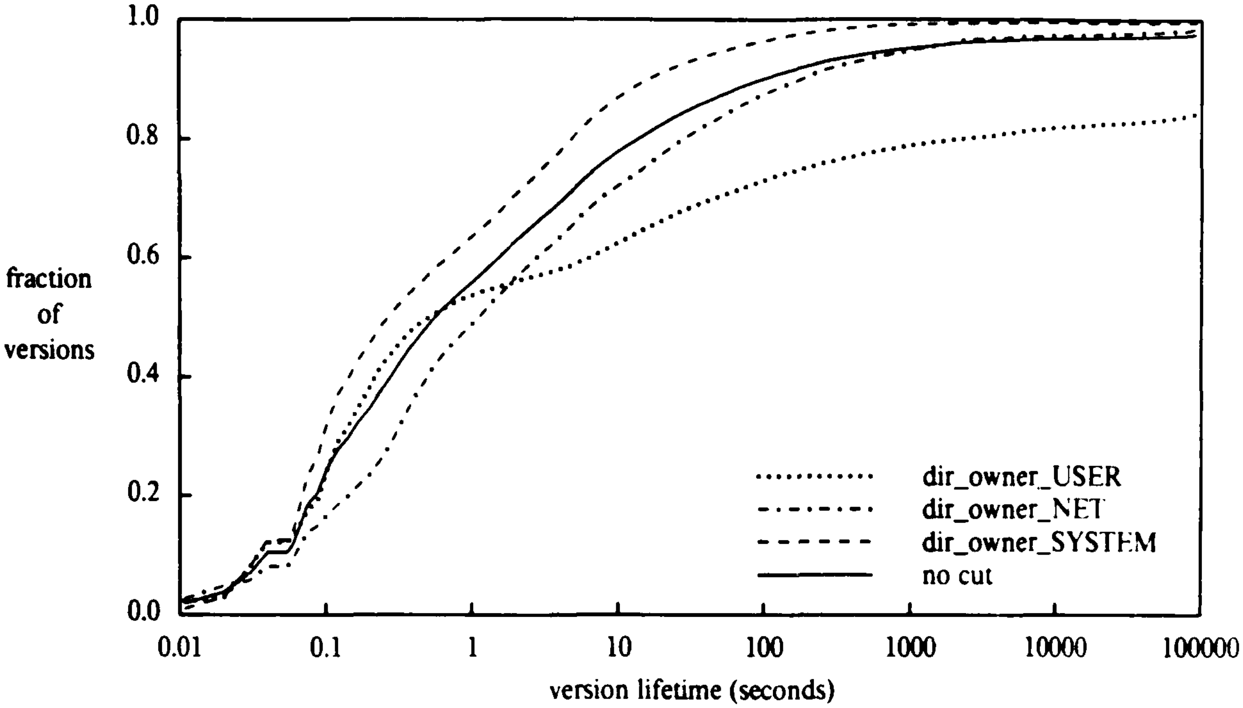


Figure 9: Directory version lifetimes (cumulative, versions living beyond log period binned at right)

Directory version lifetimes (the time from one write of a directory to the next) are shown in figure 9. Versions whose lifetime extended beyond the logging period were given infinite lifetimes (lie to the right of the histogram). Note that half of all directory versions exist for a second or less. This suggests that indiscriminate caching of referenced directories can be a mistake. This is particularly true in a DFS, where caching a remote directory may be expensive. If cached copies are flushed when updates are made, this effort will often have been wasted. Part of the reason for these short version lifetimes is heavy write activity to the system directories /tmp and /usr/spool/mqueue and to net spool directories. Most system directories and the majority of user directories remained unchanged for relatively long periods of time.

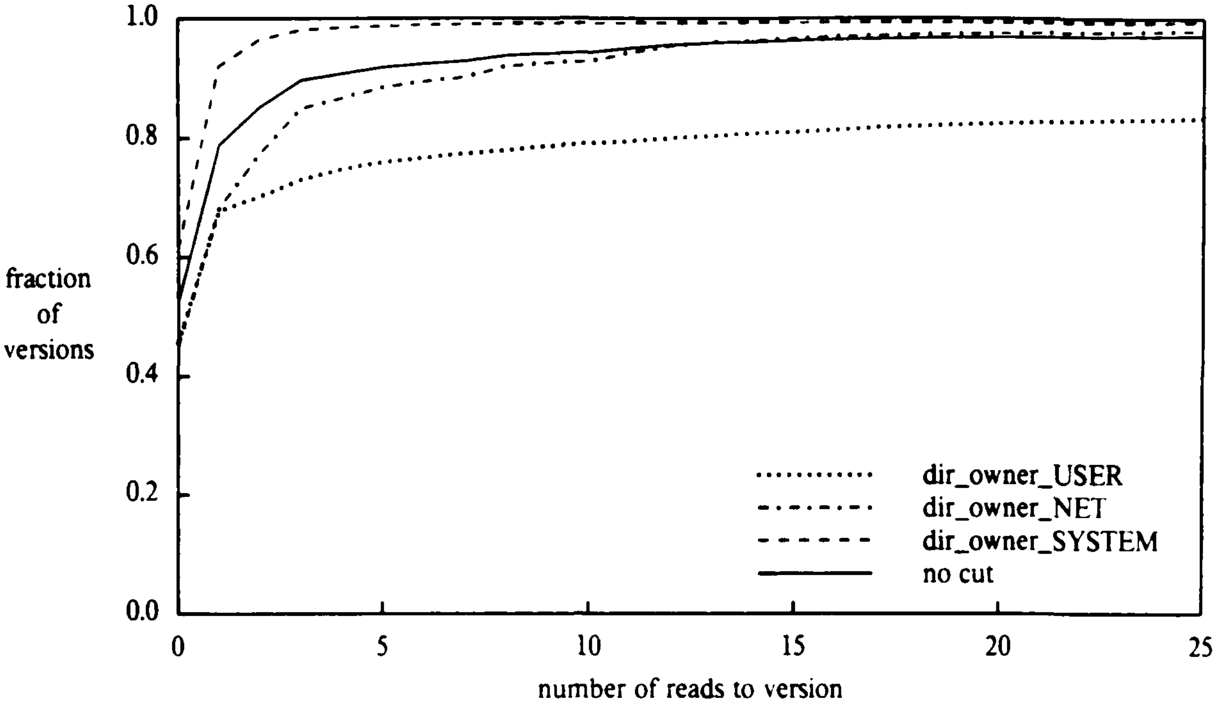


Figure 10: Reads per directory version (cumulative)

distribution	min	max	mean	median	std dev
dir_owner_NET	0	6.91e4	4.88	1	220
dir_owner_SYSTEM	0	6.17e5	23.9	0	2800
dir_owner_USER	0	3.90e3	14.2	1	74
owner_USER + ruid_USER	0	1.04e5	6.6	0	470
ruid_USER + /u	0	7.18e4	12.2	1	470
ruid_USER	0	2.38e5	11.7	0	970
no cut	0	6.17e5	14.3	0	1870

Table 13: Reads/directory version

Figure 10 and table 13 present us with another view of directory versions: the number of reads that are made to any given version. For distributed caches that discard stale copies of updated directories, this distribution can be used to estimate the number of hits one can expect to get on a cache element before it is discarded. Half of all versions are written again without being read. Roughly 4/5 of the remainder are read only a few times before being updated. While there are directory versions that can be safely cached or replicated regardless of the setup and update costs (some receive hundreds of thousands of references without being changed), separating them out from the majority of relatively useless versions may be

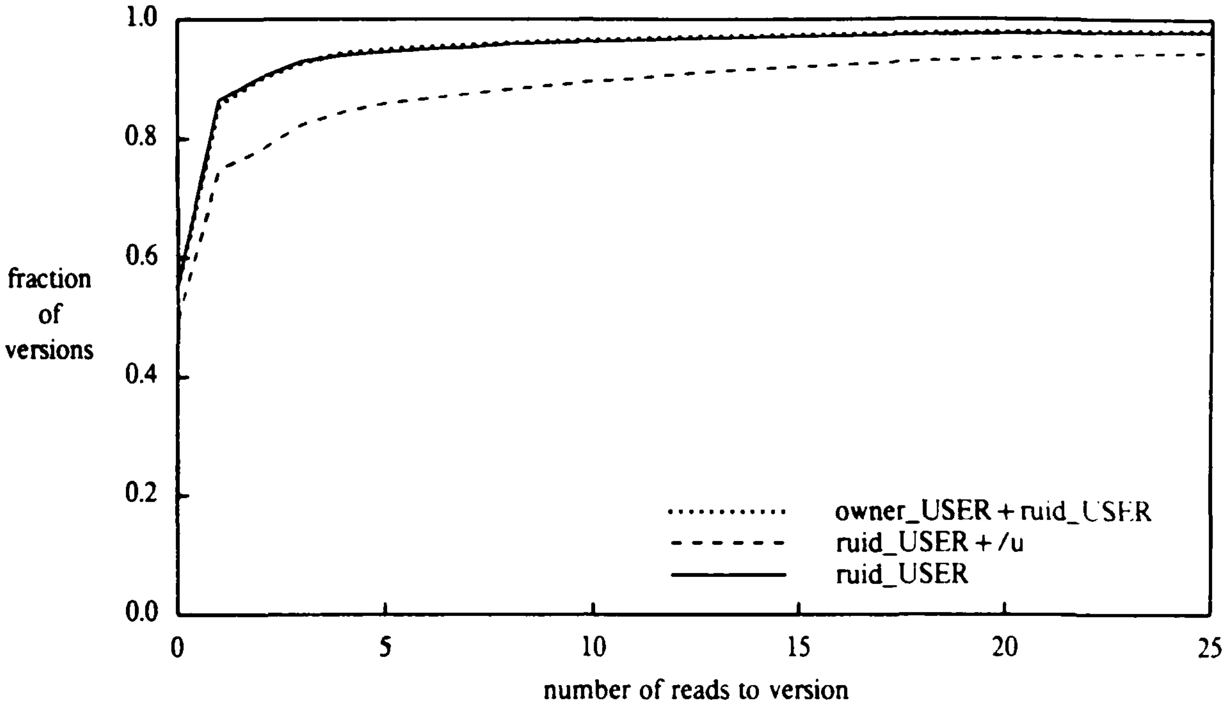


Figure 11: Reads per directory version (cumulative, user cuts)

difficult. Very cheap caching mechanisms, semantic knowledge, or knowledge of recent reference history would be useful here. For example, the knowledge that /tmp is used to store temporary files and so is frequently updated could be used to avoid potentially wasteful caching of this directory.

If we consider only user references to active directories holding user objects, we see similar distributions for reads per version (figure 11 and table 13). 86% of the versions received one or fewer user references before being updated by users. Directory versions on the /u file system saw slightly more read activity (not surprising, since heavily written system directories such as /tmp are not included here).

cut	readers		writers		users (r | w)		inversions	
	mean	>2	mean	>1	mean	>2	mean	max
dir_owner_NET	3.23	28.0%	1.35	14.6%	3.33	28.2%	34.6	6.56e3
dir_owner_SYSTEM	8.53	26.9%	0.857	3.0%	8.55	27.2%	1750	3.01e5
dir_owner_USER	1.60	9.0%	0.149	0.8%	1.60	9.0%	3.7	2.90e2
no cut	2.53	14.9%	0.487	4.2%	2.56	15.0%	149	3.01e5

Table 14: Directory sharing

cut	readers		writers		users (r | w)		inversions	
	mean	>1	mean	>1	mean	>1	mean	max
owner_USER + ruid_USER	2.50	27.9%	0.609	1.3%	2.57	28.3%	41.2	2.9e4
ruid_USER + /u	1.47	18.8%	0.351	0%	1.49	18.8%	10.8	1.3e4

Table 15: Directory sharing (user cuts)

The first two columns of table 14 show the mean number of readers per directory, as indicated by the account (ruid) of the reader, and the percentage of directories with more than *two* readers (we use two here because every directory is referenced by the housekeeping process). The next 4 columns show similar information for writers, but give the fraction with more than one writer, and for users (the overall number of distinct readers and writers). The last two columns show the mean and maximum number of *inversions* per directory. The number of inversions is the number of times that the most recent user of the directory changes (this is basically the inversion clustering metric used by Porcar [Porcar 82]). For a directory used by only one user, the number of inversions will be zero.

From table 14 we can see that 15% of the directories had multiple users (users with separate accounts). Multiple readers were much more common than multiple writers. Most of the shared directories belonged to net and system accounts. These were predominantly directories containing news articles read by many users, spool directories accessed by a number of net accounts, and directories holding widely used system files. There was relatively little sharing of user directories. Shared system directories often had a number of active users and so a high number of inversions. In a distributed environment, replication or caching of these directories would be essential.

Statistics on the sharing of active directories holding user objects are given in table 15. 1/5 of the active directories on the user file system were read by more than one user. None had multiple writers. Active directories used to resolve user objects showed a higher degree of sharing (because of shared system directories).

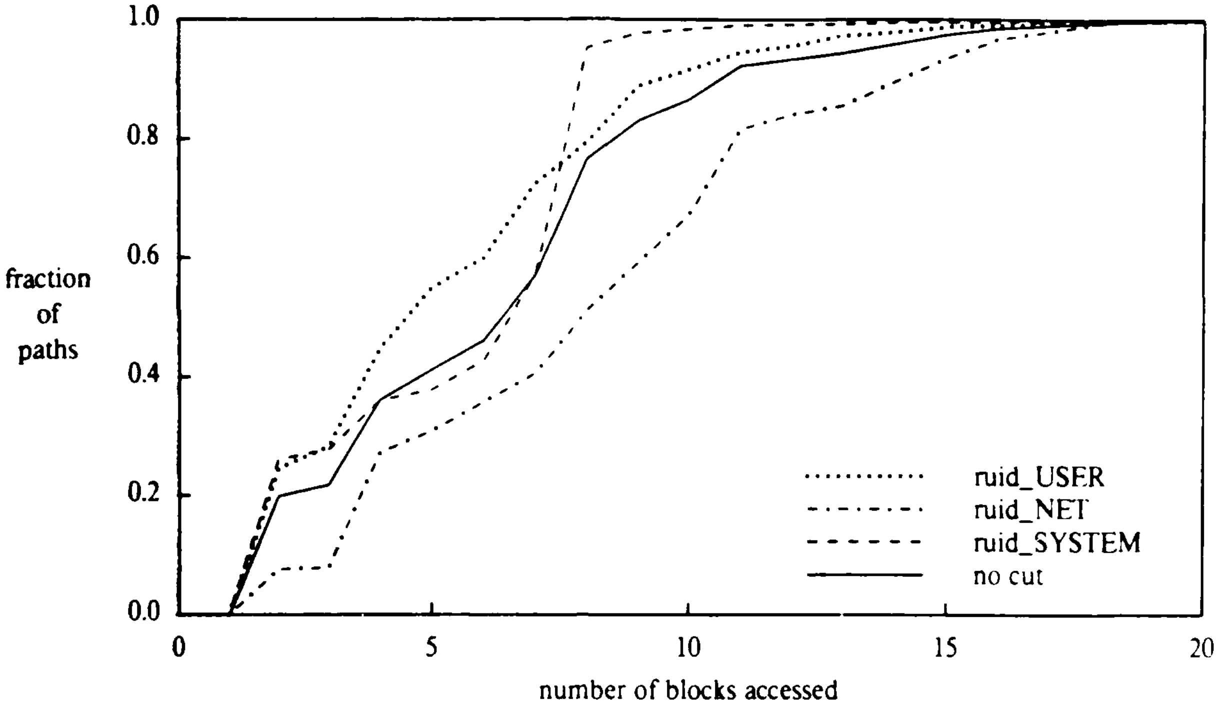


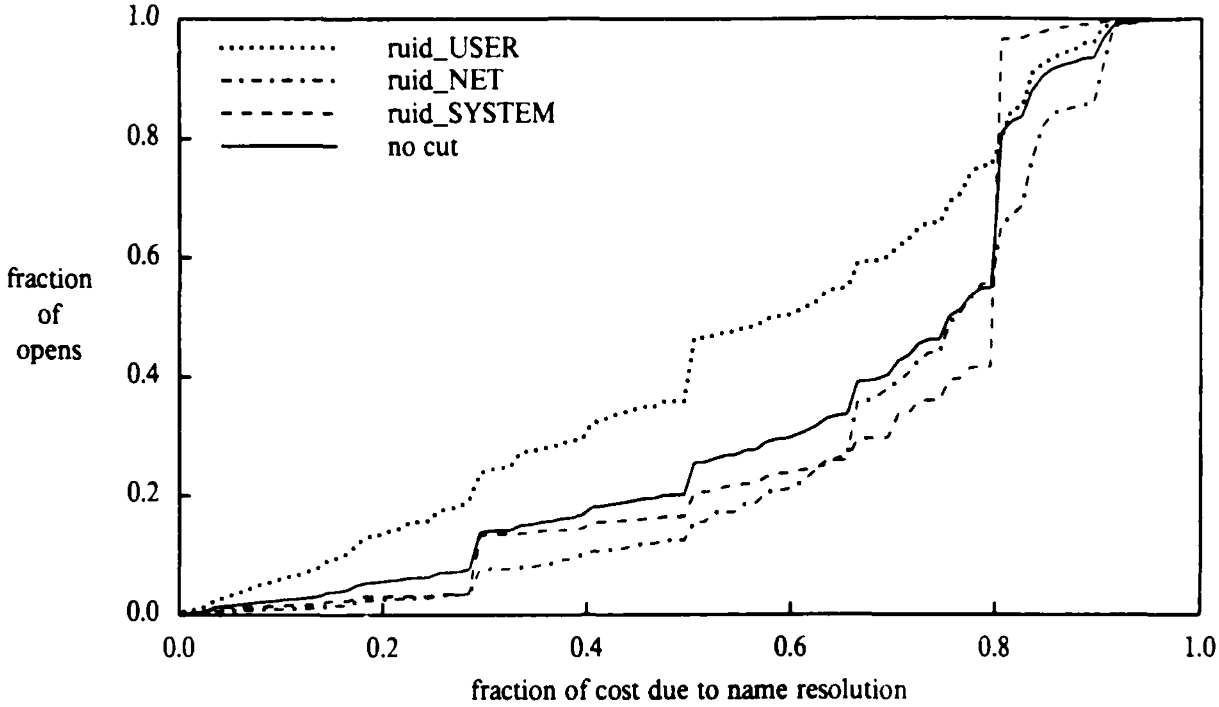
Figure 12: Path resolution cost (cumulative, 512 byte blocks)

distribution	min	max	mean	median	std dev
ruid_NET	2	22	8.43	8	4.2
ruid_SYSTEM	2	23	5.85	7	2.8
ruid_USER	2	28	5.74	5	3.4
no cut	2	28	6.61	7	3.7

Table 16: Blocks accessed/path resolution (512 byte blocks)

4.4. The High Cost of Opens

Based on the relatively small file sizes seen in studies of 4.2BSD systems and the long pathnames we have seen, it is reasonable to expect that directory overheads will be an important part of the cost of accessing a file. If we assume for the moment that no caching is done, we can estimate both the number of disk blocks that are required to resolve a path and how this compares to the number of actual file data blocks that are read or written. Reading a UNIX directory requires reading a minimum of 2 blocks: one block containing the file descriptor (inode) for the directory and at least one data block. Assuming a block size of 512 bytes, directories with no holes (empty directory entries), an average of half the entries in a directory search for a name resolution, and no caching gives the distributions shown in figure 12 and table 16. The median of 7 blocks to resolve a path is impressively large, especially when compared to the median file size of 710 bytes seen in our earlier study. Paths used by net processes are particularly long and hence expensive.



Figuie 13: Name resolution overhead for regular file opens (cumulative, 512 byte blocks)

distribution	min	max	mean	median	std dev
ruid_NET	0.0005	0.99	0.71	0.77	0.19
ruid_SYSTEM	0.002	0.98	0.68	0.81	0.20
ruid_USER	0.0001	0.99	0.55	0.59	0.26
no cut	0.0001	0.99	0.66	0.76	0.22

Table 17: Directory overhead (512 byte blocks)

Accessing file data once a path is resolved also requires a minimum of 2 blocks: one block containing the file descriptor (we ignore indirect blocks here) and at least 1 data block. If we take the ratio of the blocks required for resolving an open path to the total number of blocks required (resolution cost plus file data cost based on the amount read or written and assuming contiguous access), we get the fraction of the cost (in blocks accessed) due to the directory overhead. This is shown in figure 13 and table 17.

The directory overhead accounted for an average of 66% of the cost of accessing a regular file. This overhead accounted for the majority of the cost in 80% of the file accesses. For references made by user processes, the fraction of cost due to name resolution overhead is somewhat lower. This is due to users specifying shorter path lengths, accessing larger files, and reading a larger percentage of accessed files.

type	NONE		ruid_NET		ruid_SYSTEM		ruid_USER	
	blocks	fraction	blocks	fraction	blocks	fraction	blocks	fraction
file data	6.32e6	46.9%	1.45e6	34.1%	1.56e6	38.3%	3.30e6	66.2%
file inode	7.54e5	5.6%	2.50e5	5.9%	2.98e5	7.3%	2.06e5	4.1%
directory data	4.01e6	29.8%	1.58e6	37.2%	1.26e6	31.0%	9.94e5	20.0%
directory inode	2.39e6	17.7%	9.65e5	22.8%	9.50e5	23.3%	4.83e5	9.7%
total	1.35e7	100%	4.23e6	100%	4.07e6	100%	4.98e6	100%

Table 18: Block counts for regular file opens, reads, and writes (512 byte maximum block size)

The cost distribution weights all files equally. This give us useful information on the average overhead to access files (and so the effect of the overhead on response time), but is less useful in predicting the effect on throughput. For this we need the fraction of overall block requests that directory overhead accounts for. This information is given in table 18. Note that half of all accesses were to directory data and inode blocks.

512 bytes is a small block size by today's standards. The 4.2BSD file system on Seneca uses a block size of 4096 bytes. Figures 14 and 15 and tables 19 and 20 show what happens when we use the larger block size. The number of blocks required to resolve a path has dropped by 18%, but the fraction of cost due to the directory lookup overhead has risen sharply. It now makes up an average of 75% of the total cost and accounts for at least half of the cost in 97% of the file references. Big block sizes help most when reading file data. Directories and descriptor blocks are too small for the bigger block size to matter much.

distribution	min	max	mean	median	std dev
ruid_NET	2	16	6.93	8	3.3
ruid_SYSTEM	2	16	4.97	4	2.5
ruid_USER	2	22	4.44	4	2.5
owner_USER + ruid_USER	2	22	4.22	4	2.5
ruid_USER + /u	0	18	1.05	0	1.8
no cut	2	22	5.40	4	3.0

Table 19: Blocks accessed/path resolution (4K byte blocks)

distribution	min	max	mean	median	std dev
ruid_NET	0.004	0.99	0.80	0.84	0.11
ruid_SYSTEM	0.01	0.98	0.76	0.81	0.10
ruid_USER	0.001	0.99	0.69	0.74	0.18
owner_USER + ruid_USER	0.001	0.98	0.68	0.73	0.18
ruid_USER + /u	0	0.99	0.21	0	0.32
no cut	0.001	0.99	0.75	0.81	0.14

Table 20: Directory overhead (4K byte blocks)

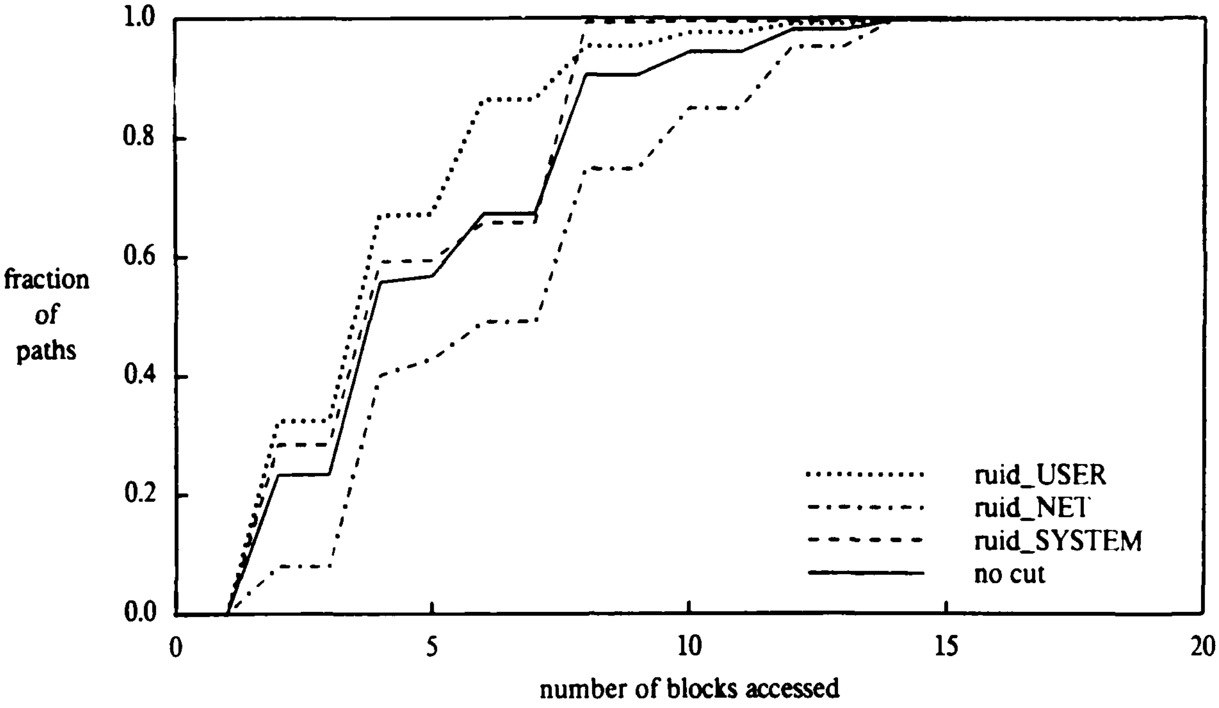


Figure 14: Path resolution cost (cumulative, 4K byte blocks)

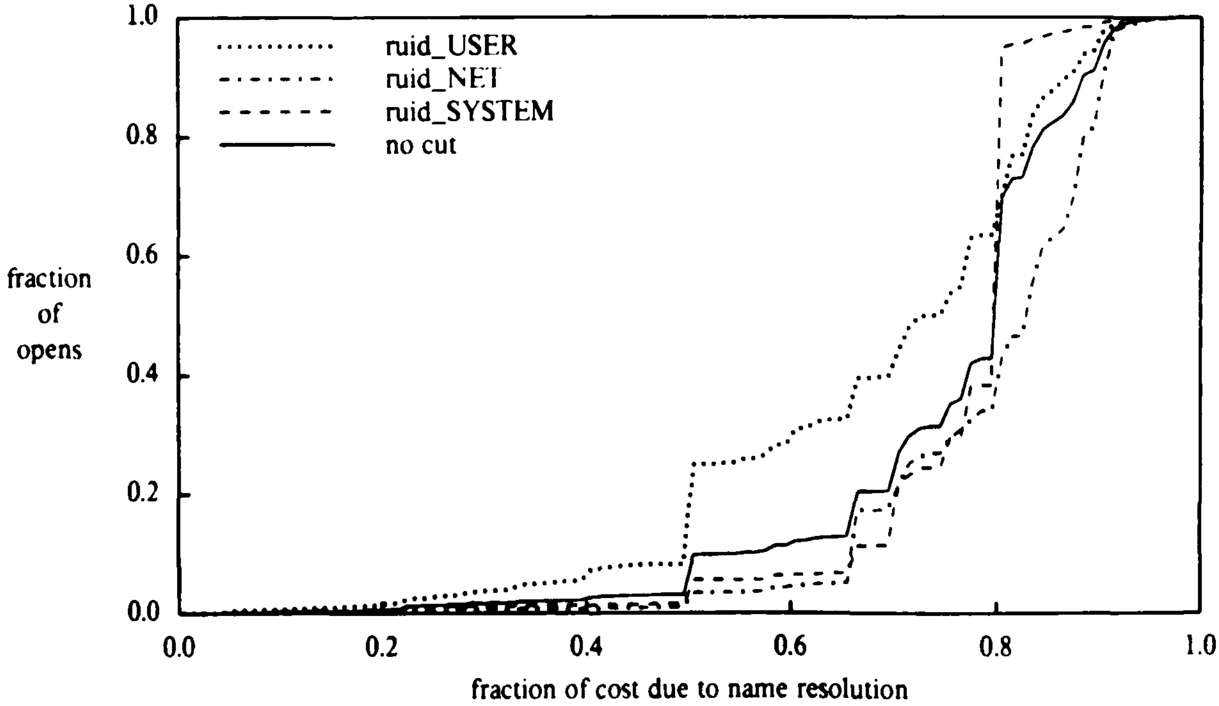


Figure 15: Name resolution overhead for regular file opens (cumulative, 4K byte blocks)

type	NONE		ruid_NET		ruid_SYSTEM		ruid_USER	
	blocks	fraction	blocks	fraction	blocks	fraction	blocks	fraction
file data	1.26e6	17.3%	3.23e5	12.1%	4.07e5	15.4%	5.27e5	28.5%
file inode	7.54e5	10.3%	2.50e5	9.4%	2.98e5	11.3%	2.06e5	11.1%
directory data	2.90e6	39.7%	1.12e6	42.2%	9.94e5	37.5%	6.36e5	34.3%
directory inode	2.39e6	32.7%	9.65e5	36.3%	9.50e5	35.9%	4.83e5	26.1%
total	7.30e6	100%	2.66e6	100%	2.6536	100%	1.85e6	100%

Table 21: Block counts for regular file opens, reads, and writes (4K byte maximum block size)

Table 21 shows the no-caching breakdown of the number of blocks of various types accessed for a 4K byte maximum block size. Note that directory data and inode blocks now account for about 3/4 of the blocks accessed. The total number of blocks accessed has been reduced to 54% of the 512 byte maximum block size figures.

Figure 16 and table 19 show the number of directory inode and data blocks accessed to resolve paths for our user cuts (assuming no caching and 4K byte maximum block sizes). Figure 17 and table 20 give the corresponding cost distributions. For the ruid_USER+/u distribution, we have included all regular file references made by user processes, but only "charged" for blocks on the /u file system. References to files on other file systems have zero resolution cost here and the cost of resolving in "/" for absolute paths is not included. Since 65% of the files referenced by user processes were actually on other file systems, the average directory overhead for this cut is low.

It should be noted that the results in this section don't apply directly to BSD UNIX. 4.2BSD maintains an extensive cache of inode, directory, and file data. 4.3BSD has added a cache of recently used directory entries. These results show, though, that "hidden costs" in UNIX file systems are significant and demonstrate how rapidly their importance increases as the block size increases.

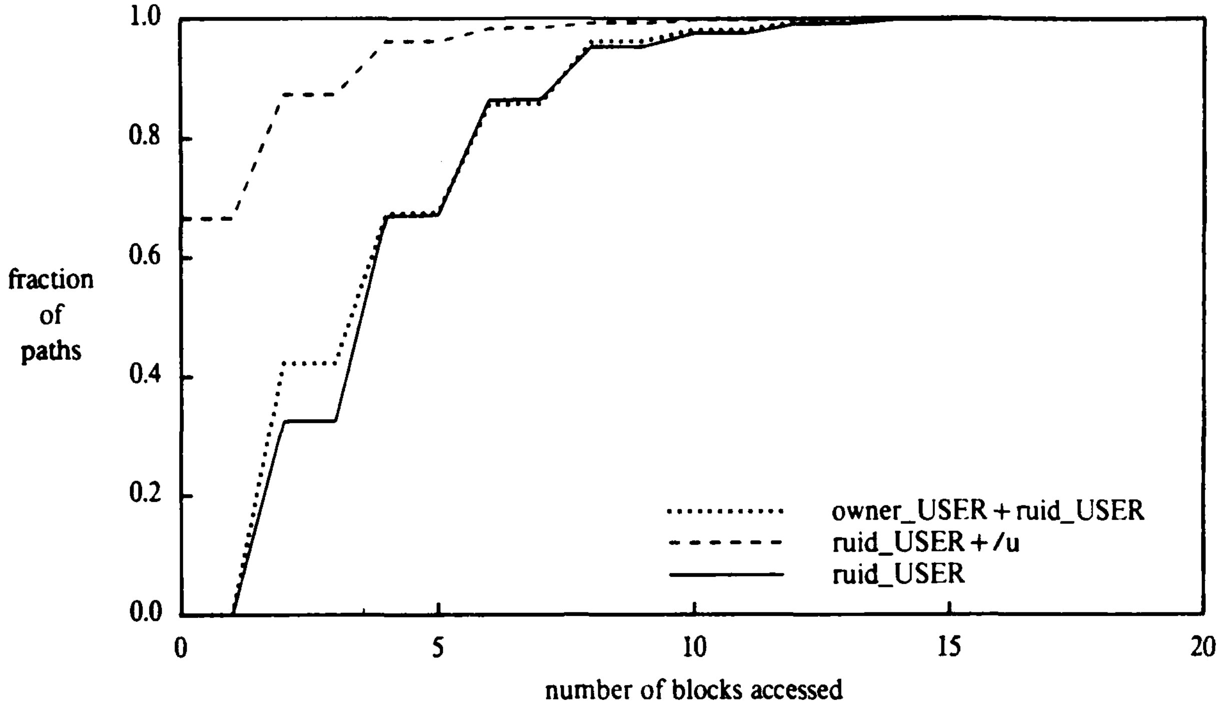


Figure 16: Path resolution cost (cumulative, 4K byte blocks, user cuts)

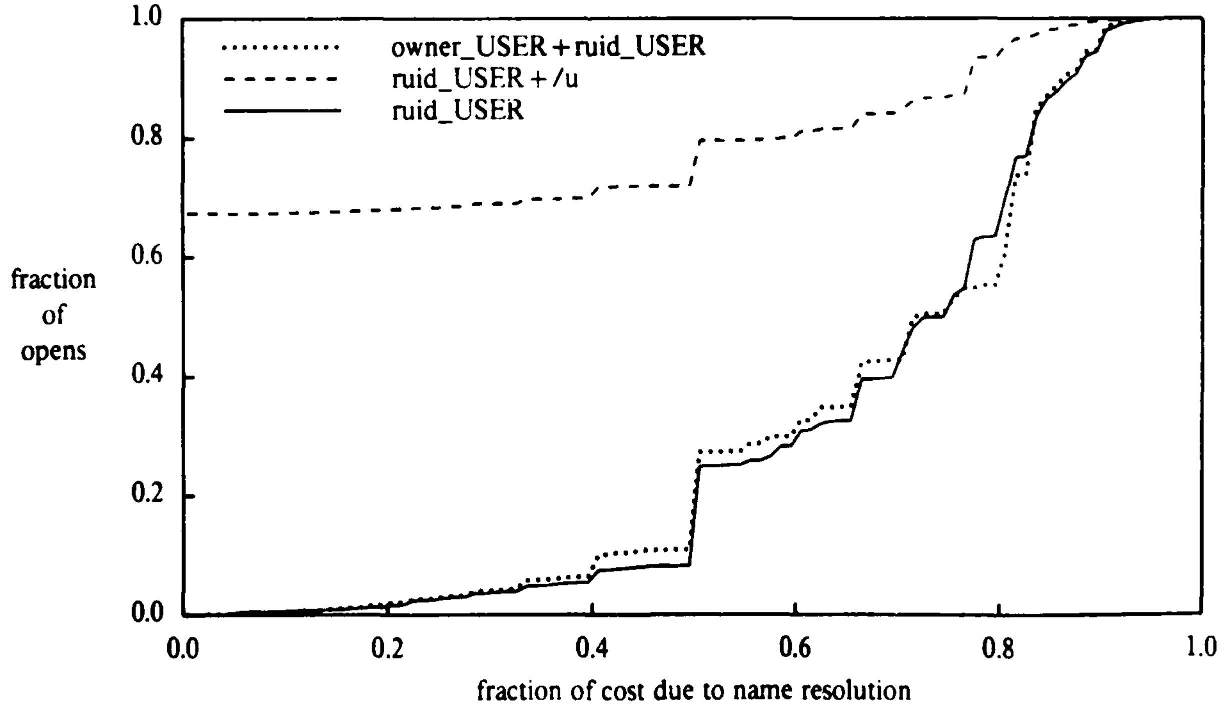


Figure 17: Name resolution overhead for file opens (cumulative, 4K byte blocks, user cuts)

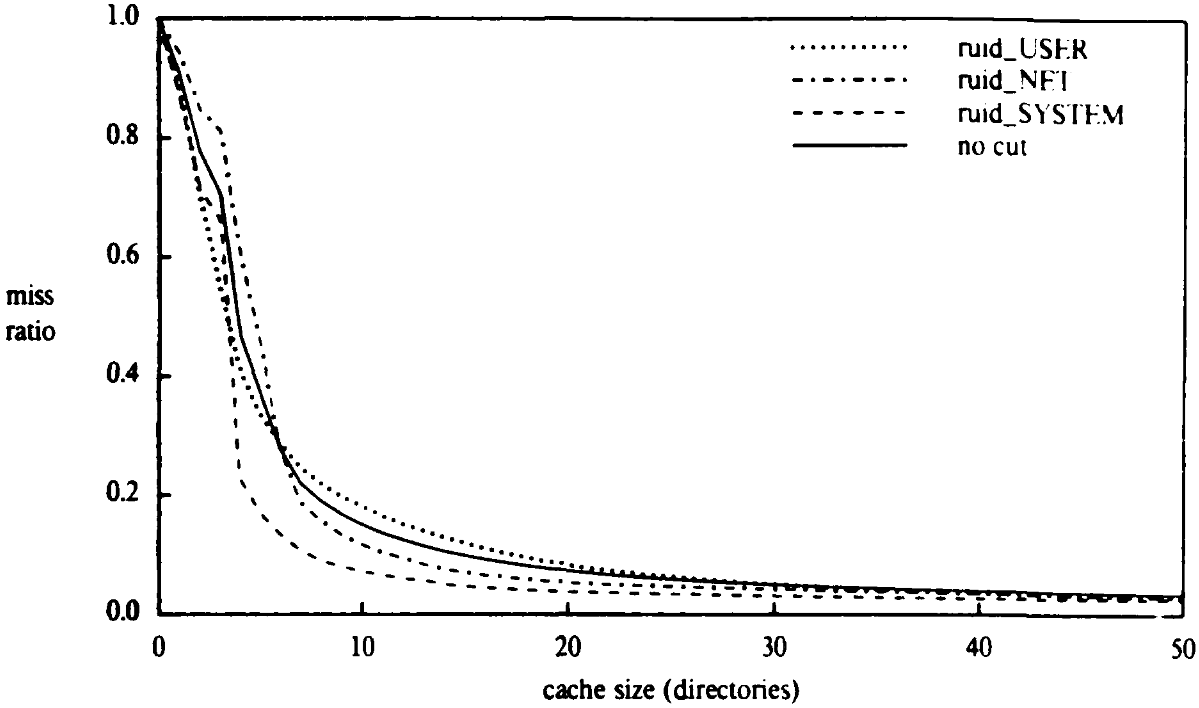


Figure 18: Whole directory cache effectiveness

nodes	no cut	ruid_NET	ruid_SYSTEM	ruid_USER	owner_USER + ruid_USER	ruid_USER + /u
5	0.37	0.46	0.17	0.34	0.25	0.14
10	0.15	0.12	0.072	0.18	0.11	0.079
15	0.099	0.070	0.045	0.12	0.076	0.065
30	0.050	0.042	0.031	0.050	0.047	0.051

Table 22: Miss ratio vs. cache size (in nodes)

4.5. Whole Directory Caching

The evidence for locality that we saw in section 4.3 suggests that a whole directory cache could be very effective in decreasing disk or net activity for name lookup. To test this idea, we simulated an LRU whole directory cache using our reference trace as input (figure 18 and table 22). We found that even a cache holding as few as 10 directory nodes achieved an 85% hit ratio. A 30 node cache gave a 95% hit ratio.

bytes	no cut	ruid_NET	ruid_SYSTEM	ruid_USER
5K	0.37	0.40	0.22	0.40
10K	0.20	0.16	0.11	0.24
15K	0.14	0.10	0.067	0.17
20K	0.10	0.079	0.052	0.13
30K	0.070	0.060	0.036	0.075
40K	0.052	0.043	0.030	0.047

Table 23: Miss ratio vs. cache size (in bytes)

User references to user objects also show a high degree of locality (figure 19 and table 22). A 10 node LRU whole directory cache captured 92% of user references to /u and 89% of references required to reach user objects.

Since directories are not generally very big, whole directory caches don't require much space (figure 20 and table 23). For the overall trace, a 14K byte cache gave the 85% hit ratio seen with the 10 node cache. Using a 41K byte cache raised the hit ratio to 95%.

As we mentioned in section 2, our log of file system activity doesn't include all directory references. Adding in the lstat and stat calls we missed could be expected to increase the effectiveness of our cache. These calls usually follow an open of the directory referencing the object of the status call and so they will all result in "hits" on the cache. Using the estimates of lstat and stat frequency made in section 3 and assuming short paths leads to a 15%-20% decrease in the miss ratios given above.

Two other studies of directory caching in UNIX environments focussed on page level caching [Sheltzer 86] and entry level caching [Leffler 84, Leffler 86]. Sheltzer et al. looked at page level caching for all references (not just our subset) on a LOCUS system [Walker 83] (an enhanced, distributed version of 4.1BSD). Their simulations assume, though, that directories fit in a single page. While this is true for most directories on a BSD UNIX system, there are a few large heavily used system directories that typically contain in excess of 100 entries (see figure 3). These directories are referenced frequently enough to make the high hit ratios found by Sheltzer et al. questionable for page sizes one might typically use with directories (1K bytes or less because of the small size of directories). As figure 3 shows, inferring dynamic distributions from static ones can be dangerous. Their simulations are actually, then, for a whole directory LRU cache. Their result of hit ratios of 87%-96% (depending on the system) for a 40 page (node) cache agrees with our result of a 96% hit ratio for a 40 node cache.

Leffler et al., in tuning and enhancing 4.2BSD, found that a system wide entry level cache containing 400 entries (about 18K bytes) gave a 60% hit ratio. This was coupled with a per process directory offset cache having a 25% hit ratio (to catch directory scans), giving an overall hit ratio of 85%. This is effectively an entry cache and a per process single directory cache, with invalidation on update. We saw a hit ratio of 89% for an 18K byte whole directory cache. Our hit ratios are higher because we effectively "read ahead" for processes scanning directories by caching the entire node on 1st reference, don't need to invalidate on update or working directory changes, and cache globally. Maintenance and/or search costs may be a problem for whole directory caches, though. Per entry caches are comparatively easy to search and maintain.

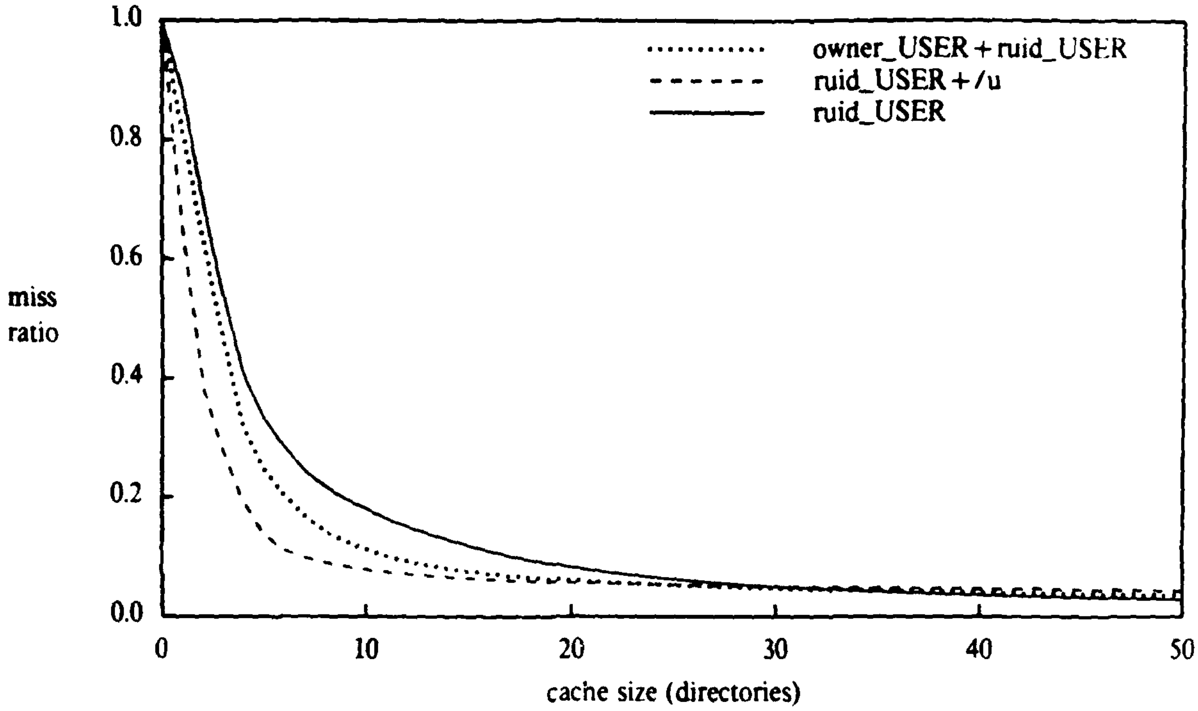


Figure 19: Whole directory cache effectiveness (user cuts)

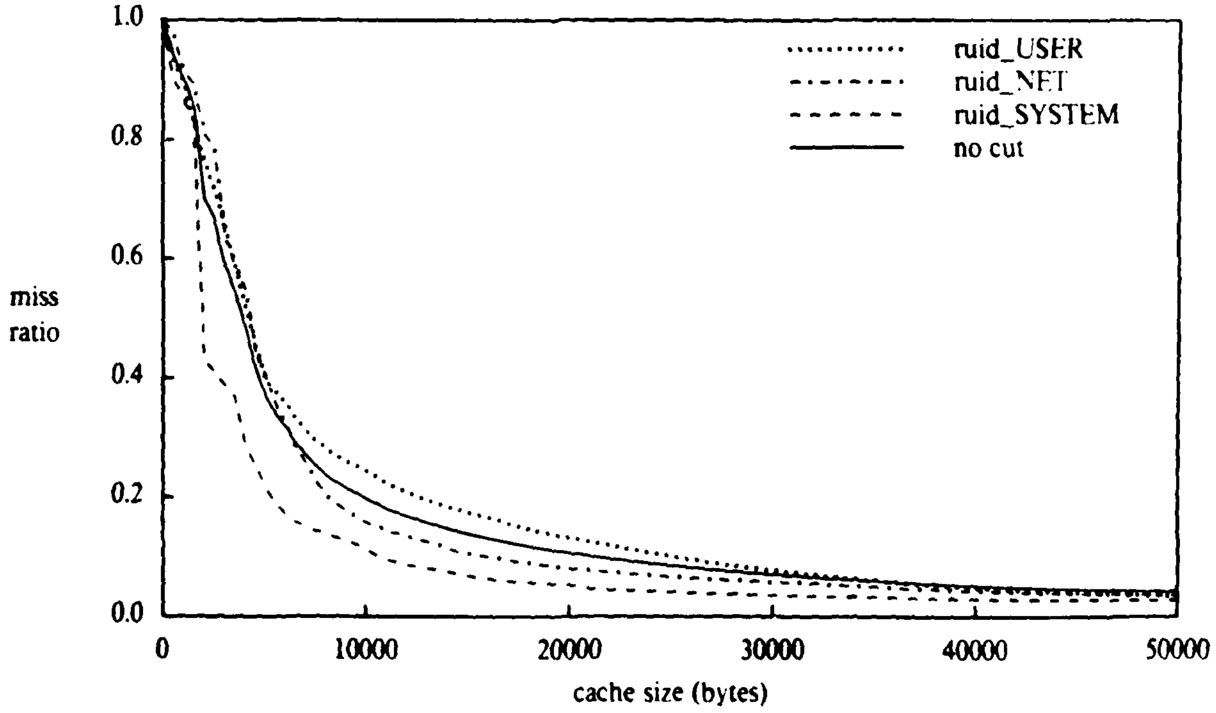


Figure 20: Whole directory cache effectiveness (byte size limit)

5. Implications

In this section we make some observations on local and distributed file system design, based on the results we have presented. It should be emphasized that these observations and suggestions are most applicable to systems that see reference patterns similar to ours. They will not necessarily carry over to other environments.

We found that 70% of the logged paths were absolute references. This implies that deep directory trees raise the cost of references. For distributed file systems in particular, the root of the file system must be cheap to access, since it will be heavily used.

Nearly 3/4 of all references went to system directories. Less than 10% went to user directories. For DFS's that support access to local file systems coupled with access to a global user file system, minimizing the performance impact of adding the global file system on local accesses is clearly important. Conversely, carefully coupling transparent access to a network file system holding user files with cheap access to local files can result in a coherent distributed file system with good overall performance (at least for name resolution), even in situations where net access is expensive.

93.4% of references were for read in our data (the actual overall figure for the system was probably somewhat higher). Clearly most directories should be optimized for lookup. Some directories are heavily written and rarely read, though. Other organizations may be appropriate for these directories. The use of semantic information or past history would be useful here.

Most directory versions were short lived (over half lasted less than a second). There is no point in writing these versions back to disk and so delaying writes of directories will improve performance (although not, perhaps, reliability).

Most directory versions received only a few references. This, combined with the short lifetime of most versions, implies that caching these versions serves no purpose.

Directory inter-reference times were short (half were 1/4 second or less). Any strategy that attempts to decrease the overhead for the next reference will have to act quickly. Migrating or caching at the time of first access looks attractive here.

Large block sizes help when reading some files, but many files, most directories and all descriptor blocks are too small to benefit from larger block sizes. Coupling larger block sizes with a file system design that ties file descriptors, directories, and data together on disk could be expected to lead to substantial performance improvements. The 4.2BSD file system does this by attempting to allocate these related items close together on disk [McKusick 84]. Further improvement (at the cost of increased crash recovery complexity) could be had by allocating inodes and at least the initial data in files contiguously on disk. On a 4K block size file system, this could be expected to lead to about a 40% decrease in transfers for file open, read, and write activity (table 21), given equivalent caching effectiveness for data, inodes and directories.

The combination of relatively long path names and small files means that, in the absence of caching, the majority of UNIX file system activity is in support of name resolution. Optimizations in this area are likely to produce significant performance improvements, particularly as the block size used for data transfers increases.

Most references go to a handful of directories. Our studies show that, because of this locality, a small cache gives good results. Since there are few changes to these directories, replication would be a cheap alternative in a DFS.

Entry level caches, by themselves, are much less effective than node level caches because of the frequent sequential access of entries in a directory. Caching strategies that recognize and exploit this sequentiality will do well.

6. Further Work

As we have shown here, the analysis of file system traces can soak up boundless amounts of time and energy. We have tried to stop at the point where we felt that we had enough information to understand trace driven simulations based on the data. There is a great deal of related work that could be done. Some possibilities include:

(1) Further data collection and analysis for different environments and work loads. This would give us a better feeling for where our data fits into the universe of file system usage.

(2) Examining in more detail the activity per user. This information would be useful in designing DFS's that include personal workstations.

(3) Fitting curves to various distributions (inter-reference time, cache miss ratio, and so on). These would be useful in writing synthetic drivers for use in simulating file systems [Satyanarayanan 83].

(4) Using the trace data to drive other simulations investigating file system performance and caching issues. A trace driven simulation of Roe is planned [Floyd 87].

(5) Investigating the correlation between directory depth and activity. Since most paths are absolute, one would expect that directories close to the root of the directory tree will be referenced more frequently than those towards the leaves.

(6) Simulations of the directory reference overhead cost taking into account cache hits.

(7) Most files and directories on 4.2BSD systems are small. Given this, it is not clear that storing inodes separately is a good idea. A redesign of the UNIX file system to store inodes and data together on disk (at least for the common case of small objects) could result in substantial performance increases. The 4.2BSD file system takes steps in this direction. How much better can we do?

(8) Investigating other directory organizations. As block sizes increase, the directory organization used by UNIX becomes less and less appropriate. "Cheaper" ways of representing and using hierarchical directories are needed.

7. Summary

This paper has analyzed in some detail directory reference patterns resulting from primarily open activity on a 4.2BSD UNIX system supporting university research. Our major findings:

(1) Directories are mostly small, with half holding under 8 entries. Referenced directories are somewhat larger (median of 26 entries).

(2) 3/4 of all references are made to system directories. Most references go to a few very active system directories.

(3) Reads account for 93.4% of the references we see and writes for 6.6% of the references.

(4) 70% of all paths are specified absolutely. Relatively few paths (26%) reference objects in current working directories.

(5) Paths are "long", having an average of 2.70 components.

(6) Inter-reference times are short. Half are 1/4 second or less.

(7) Directory versions are usually sort lived (half live less than a second) and receive few reads.

(8) The combination of small file sizes and long access paths means that name resolution overhead is high. In the absence of caching and using a 4K byte maximum block size, 72% of the blocks accessed in opening and using files are for name resolution.

(9) There is a high degree of locality in directory references. A 10 node (14K byte) cache achieves an 85% hit ratio and a 30 node (41K byte) cache has a 95% hit ratio.

Overall, our results show that hierarchical directories can be expensive, but that there is a high degree of locality in reference patterns. This locality allows the use of caching to dramatically reduce the expense of directory lookup.

As is true with all studies of this sort, our results can be guaranteed to be valid only for our system at the time of data collection. Care should be taken in applying the results to other situations.

8. Acknowledgements

Carla Ellis and Stuart Friedberg made a number of useful suggestions on the analysis and presentation. Both their help and patience are gratefully acknowledged. Speculations by Ousterhout et al. [Ousterhout 85] on 4.2BSD directory overheads helped inspire parts of this study. Jeff Mogul convinced me that I really should do the whole directory caching studies that I kept avoiding. Lee Moore's efforts in maintaining and enhancing our press software [Kahrs 85] made the plots shown here possible. Finally, I would like to thank the 4 VAXen that worked so hard to produce the results shown here and to commemorate the one that died trying.

References

[Ellis 83] Ellis, C. and Floyd, R., "The Roe File System," *Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems*, October 1983, 175-81.

[Floyd 85] Floyd, R. A., "Short Term File Reference Patterns in a UNIX Environment: Preliminary Results," Internal Note, Department of Computer Science, University of Rochester, August 1985.

[Floyd 86] Floyd, R. A., "Short-Term File Reference Patterns in a UNIX Environment," TR 177, Department of Computer Science, University of Rochester, March 1986.

[Floyd 87] Floyd, R. A., *Transparency in Distributed File Systems*, Ph.D. Dissertation, Department of Computer Science, University of Rochester, February 1987. (in preparation).

[Kahrs 85] Kahrs, M. and Moore, L., "Adventures with Typesetter-Independent TROFF," Technical Report 159, Department of Computer Science, University of Rochester, June 1985.

[Leffler 84] Leffler, S., Karels, M. and McKusick, M., "Measuring and Improving the Performance of 4.2BSD," *1984 USENIX Summer Conference Proceedings*, June 1984, 237-52.

[Leffler 86] Leffler, S., Private Communication, August 1986.

[McKusick 84] McKusick, M., Joy, W., Leffler, S. and Fabry, R., "A Fast File System For UNIX," *ACM Transactions on Computer Systems 2*:3, August 1984, 181-197.

[Mogul 86a] Mogul, J., Private Communication, July 1986.

[Mogul 86b] Mogul, J., "Representing Information about Files," STAN-CS-86-1103, Ph.D. Dissertation, Department of Computer Science, Stanford University, March 1986.

[Nowitz 78] Nowitz, D. and Lesk, M., "A Dial-Up Network of UNIX Systems," in *The UNIX Programmer's Manual, Seventh Edition*, vol. 2, Bell Laboratories, August 1978.

[Ousterhout 85] Ousterhout, J., Da Costa, H., Harrison, D., Kunze, J., Kupfer, M. and Thompson, J., "A Trace Driven Analysis of the UNIX 4.2BSD File System," UCB/Computer Science Department 85/230, EECS Department, University of California, Berkeley, April 1985.

[Porcar 82] Porcar, J., "File Migration in Distributed Computer Systems," LBL-14763, Lawrence Berkeley Laboratory, July 1982.

[Ritchie 78] Ritchie, D. and Thompson, K., "The UNIX Time-Sharing System," *Bell System Technical Journal 57*:6, Part 2, July-August 1978, 1905-30.

[Satyanarayanan 83] Satyanarayanan, M., "A Methodology for Modelling Storage Systems and its Application to a Network File System," CMU-CS-83-109, Department of Computer Science, Carnegie-Mellon University, March 1983.

[Satyanarayanan 85] Satyanarayanan, M., Howard, J., Hichols, D., Sidebotham, R., Spector, A. and West, M., "The ITC Distributed File System: Principles and Design," *Operating Systems Review 19*:5, December

1985. 35-50. (SOSP 10).

[Sheltzer 86] Sheltzer. A.. Lindell. R. and Popek. G., "Name Service Locality and Cache Design in a Distributed Operating System." *Proceedings of the 6th International Conference on Distributed Computing Systems*. May 1986. 515-522.

[Tichy 84] Tichy. W. and Zuwang. R., "Towards a Distributed File System," *1984 USENIX Summer Conference Proceedings*. June 1984. 87-97.

[Walker 83] Walker. B.. Popek. G.. English. R.. Kline. C. and Thiel. G.. "The LOCUS Distributed Operating System," *Operating Systems Review 17*:5. December 1983. 49-70. (SOSP 9).

END

5_87

DTIC